A Formalization of Declassification with WHAT&WHERE-Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

April 23, 2014

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private sources to public sinks. Noninterference captures this intuition by requiring that no information whatsoever flows from private sources to public sinks. However, in practice this definition is often too strict: Depending on the intuitive desired security policy, the controlled declassification of certain private information (WHAT) at certain points in the program (WHERE) might not result in an undesired information leak.

We present an Isabelle/HOL formalization of such a security property for controlled declassification, namely WHAT&WHERE-security from [2]. The formalization includes compositionality proofs for and a soundness proof for a security type system that checks for programs in a simple while language with dynamic thread creation.

Our formalization of the security type system is abstract in the language for expressions and in the semantic side conditions for expressions. It can easily be instantiated with different syntactic approximations for these side conditions. The soundness proof of such an instantiation boils down to showing that these syntactic approximations imply the semantic side conditions.

This Isabelle/HOL formalization uses theories from the entry Strong-Security (see proof document for details).

Contents

1 Preliminary definitions

1.1 Type synonyms

The formalization is parametric in different aspects. Notably, it is parametric in the security lattice it supports.

For better readability, we use the following type synonyms in our formalization (from the entry Strong-Security):

theory *Types*
imports *Main*
begin

— type parameters:
— 'exp: expressions (arithmetic, boolean...)
— 'val: values
— 'id: identifier names
— 'com: commands
— 'd: domains

This is a collection of type synonyms. Note that not all of these type synonyms are used within Strong-Security - some are used in WHATandWHERE-Security.

— type for memory states - map ids to values
type-synonym $('id,\ 'val)\ State = 'id \Rightarrow 'val$

— type for evaluation functions mapping expressions to a values depending on a state
type-synonym $('exp,\ 'id,\ 'val)\ Evalfunction =$
$'exp \Rightarrow ('id,\ 'val)\ State \Rightarrow 'val$

— define configurations with threads as pair of commands and states
type-synonym $('id,\ 'val,\ 'com)\ TConfig = 'com \times ('id,\ 'val)\ State$

— define configurations with thread pools as pair of command lists (thread pool) and states

type-synonym $('id, 'val, 'com)$ *TPConfig* $=$
$('com\ list) \times ('id, 'val)\ State$

— type for program states (including the set of commands and a symbol for terminating - None)
type-synonym $'com\ ProgramState = 'com\ option$

— type for configurations with program states
type-synonym $('id, 'val, 'com)$ *PSConfig* $=$
$'com\ ProgramState \times ('id, 'val)\ State$

— type for labels with a list of spawned threads
type-synonym $'com\ Label = 'com\ list$

— type for step relations from single commands to a program state, with a label
type-synonym $('exp, 'id, 'val, 'com)$ *TLSteps* $=$
$(('id, 'val, 'com)\ TConfig \times 'com\ Label$
$\times ('id, 'val, 'com)\ PSConfig)\ set$

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)$ *TLSteps-curry* $=$
$'com \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ Label \Rightarrow 'com\ ProgramState$
$\Rightarrow ('id, 'val)\ State \Rightarrow bool$

— type for step relations from thread pools to thread pools
type-synonym $('exp, 'id, 'val, 'com)$ *TPSteps* $=$
$(('id, 'val, 'com)\ TPConfig \times ('id, 'val, 'com)\ TPConfig)\ set$

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)$ *TPSteps-curry* $=$
$'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow bool$

— define type of step relations for single threads to thread pools
type-synonym $('exp, 'id, 'val, 'com)$ *TSteps* $=$
$(('id, 'val, 'com)\ TConfig \times ('id, 'val, 'com)\ TPConfig)\ set$

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)
type-synonym $('exp, 'id, 'val, 'com)$ *TSteps-curry* $=$
$'com \Rightarrow ('id, 'val)\ State \Rightarrow 'com\ list \Rightarrow ('id, 'val)\ State \Rightarrow bool$

— type for simple domain assignments; 'd has to be an instance of order (partial order
type-synonym $('id, 'd)$ *DomainAssignment* $= 'id \Rightarrow 'd{::}order$

type-synonym $'com$ *Bisimulation-type* $= (('com\ list) \times ('com\ list))\ set$

— type for escape hatches
type-synonym $('d, 'exp)\ Hatch = 'd \times 'exp$

— type for sets of escape hatches
type-synonym $('d,\ 'exp)$ *Hatches* $= (('d,\ 'exp)\ Hatch)\ set$

— type for local escape hatches
type-synonym $('d,\ 'exp)$ *lHatch* $= 'd \times 'exp \times nat$

— type for sets of local escape hatches
type-synonym $('d,\ 'exp)$ *lHatches* $= (('d,\ 'exp)\ lHatch)\ set$

end

2 WHAT&WHERE-security

2.1 Definition of WHAT&WHERE-security

The definition of WHAT&WHERE-security is parametric in a security lattice (*'d*) and in a programming language (*'com*).

theory *WHATWHERE-Security*
imports *../Strong−Security/Types*
begin

locale *WHATWHERE* =
fixes $SR :: ('exp,\ 'id,\ 'val,\ 'com)\ TLSteps$
and $E :: ('exp,\ 'id,\ 'val)\ Evalfunction$
and $pp :: 'com \Rightarrow nat$
and $DA :: ('id,\ 'd::order)\ DomainAssignment$
and $lH :: ('d::order,\ 'exp)\ lHatches$

begin

— define when two states are indistinguishable for an observer on domain d
definition *d-equal* $:: 'd::order \Rightarrow ('id,\ 'val)\ State$
  $\Rightarrow ('id,\ 'val)\ State \Rightarrow bool$
where
d-equal $d\ m\ m' \equiv \forall x.\ ((DA\ x) \leq d \longrightarrow (m\ x) = (m'\ x))$

abbreviation *d-equal'* $:: ('id,\ 'val)\ State$
  $\Rightarrow 'd::order \Rightarrow ('id,\ 'val)\ State \Rightarrow bool$
((- =$_-$ -))
where
$m =_d m' \equiv$ *d-equal* $d\ m\ m'$

— transitivity of d-equality
lemma *d-equal-trans*:

$[\![\ m =_d m'; m' =_d m''\ ]\!] \Longrightarrow m =_d m''$
by (*simp add*: *d-equal-def*)

abbreviation *SRabbr* :: (*'exp*, *'id*, *'val*, *'com*) *TLSteps-curry*
((*1* ⟨-,/-⟩) →◁-▷/ (*1* ⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
$\langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle \equiv ((c,m),\alpha,(p,m')) \in SR$

— function for obtaining the unique memory (state) after one step for a command and a memory (state)
definition *NextMem* :: *'com* ⇒ (*'id*, *'val*) *State* ⇒ (*'id*, *'val*) *State*
(⟦-⟧'(-'))
where
$[\![c]\!](m) \equiv (THE\ m'.\ (\exists\, p\ \alpha.\ \langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle))$

— function getting all escape hatches for some location
definition *htchLoc* :: *nat* ⇒ (*'d*, *'exp*) *Hatches*
where
$htchLoc\ \iota \equiv \{(d,e).\ (d,e,\iota) \in lH\}$

— function for getting all escape hatches for some set of locations
definition *htchLocSet* :: *nat set* ⇒ (*'d*, *'exp*) *Hatches*
where
$htchLocSet\ PP \equiv \bigcup \{h.\ (\exists\, \iota \in PP.\ h = htchLoc\ \iota)\}$

— predicate for (d,H)-equality
definition *dH-equal* :: *'d* ⇒ (*'d*, *'exp*) *Hatches*
⇒ (*'id*, *'val*) *State* ⇒ (*'id*, *'val*) *State* ⇒ *bool*
where
$dH\text{-}equal\ d\ H\ m\ m' \equiv (m =_d m' \wedge$
$(\forall\, (d',e) \in H.\ (d' \leq d \longrightarrow (E\ e\ m = E\ e\ m'))))$

abbreviation *dH-equal'* :: (*'id*, *'val*) *State* ⇒ *'d* ⇒ (*'d*, *'exp*) *Hatches*
⇒ (*'id*, *'val*) *State* ⇒ *bool*
((- $\sim_{-,-}$ -))
where
$m \sim_{d,H} m' \equiv dH\text{-}equal\ d\ H\ m\ m'$

— predicate indicating that a command is not a d-declassification command
definition *NDC* :: *'d* ⇒ *'com* ⇒ *bool*
where
$NDC\ d\ c \equiv (\forall\, m\ m'.\ m =_d m' \longrightarrow [\![c]\!](m) =_d [\![c]\!](m'))$

— predicate indicating an 'immediate d-declassification command' for a set of escape hatches
definition *IDC* :: *'d* ⇒ *'com* ⇒ (*'d*, *'exp*) *Hatches* ⇒ *bool*
where

$IDC\ d\ c\ H \equiv (\exists\, m\ m'.\ m =_d m' \wedge$
$(\neg\ [\![c]\!](m) =_d [\![c]\!](m')))$
$\wedge\ (\forall\, m\ m'.\ m \sim_{d,H} m' \longrightarrow [\![c]\!](m) =_d [\![c]\!](m')\ )$

definition *stepResultsinR* :: $'com\ ProgramState \Rightarrow\ 'com\ ProgramState$
$\Rightarrow\ 'com\ Bisimulation\text{-}type \Rightarrow bool$
where
$stepResultsinR\ p\ p'\ R \equiv (p = None \wedge p' = None) \vee$
$(\exists\, c\ c'.\ (p = Some\ c \wedge p' = Some\ c' \wedge ([c],[c']) \in R))$

definition *dhequality-alternative* :: $'d \Rightarrow nat\ set \Rightarrow nat$
$\Rightarrow ('id,\ 'val)\ State \Rightarrow ('id,\ 'val)\ State \Rightarrow bool$
where
$dhequality\text{-}alternative\ d\ PP\ \iota\ m\ m' \equiv m \sim_{d,(htchLocSet\ PP)} m' \vee$
$(\neg\ (htchLoc\ \iota) \subseteq (htchLocSet\ PP))$

definition *Strong-dlHPP-Bisimulation* :: $'d \Rightarrow nat\ set$
$\Rightarrow\ 'com\ Bisimulation\text{-}type \Rightarrow bool$
where
$Strong\text{-}dlHPP\text{-}Bisimulation\ d\ PP\ R \equiv$
$(sym\ R) \wedge (trans\ R) \wedge$
$(\forall\, (V,V') \in R.\ length\ V = length\ V') \wedge$
$(\forall\, (V,V') \in R.\ \forall\, i < length\ V.$
$((NDC\ d\ (V!i)) \vee$
$(IDC\ d\ (V!i)\ (htchLoc\ (pp\ (V!i)))))) \wedge$
$(\forall\, (V,V') \in R.\ \forall\, i < length\ V.\ \forall\, m1\ m1'\ m2\ \alpha\ p.$
$(\ \langle V!i,m1\rangle \rightarrow\lhd\alpha\rhd\ \langle p,m2\rangle \wedge m1 \sim_{d,(htchLocSet\ PP)} m1')$
$\longrightarrow (\exists\, p'\ \alpha'\ m2'.\ (\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle \wedge$
$(stepResultsinR\ p\ p'\ R) \wedge (\alpha,\alpha') \in R \wedge$
$(dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'))))$

— predicate to define when a program is strongly secure
definition *WHATWHERE-Secure* :: $'com\ list \Rightarrow bool$
where
$WHATWHERE\text{-}Secure\ V \equiv (\forall\, d\ PP.$
$(\exists\, R.\ Strong\text{-}dlHPP\text{-}Bisimulation\ d\ PP\ R \wedge (V,V) \in R))$

— auxiliary lemma to obtain central strong (d,lH,PP)-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)
lemma *strongdlHPPB-aux*:
$\bigwedge V\ V'\ m1\ m1'\ m2\ p\ i\ \alpha.\ [\![\ Strong\text{-}dlHPP\text{-}Bisimulation\ d\ PP\ R;$
$i < length\ V;\ (V,V') \in R;$
$\langle V!i,m1\rangle \rightarrow\lhd\alpha\rhd\ \langle p,m2\rangle;\ m1 \sim_{d,(htchLocSet\ PP)} m1'\ ]\!]$
$\Longrightarrow (\exists\, p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle$
$\wedge\ stepResultsinR\ p\ p'\ R \wedge (\alpha,\alpha') \in R \wedge$
$(dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'))$

by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *fastforce*)

— auxiliary lemma to obtain 'NDC or IDC' from strong (d,lH,PP)-Bisimulation as lemma in meta logic allowing instantiation of the variables
lemma *strongdlHPPB-NDCIDCaux*:
$\bigwedge V\ V'\ i.\ [\![Strong\text{-}dlHPP\text{-}Bisimulation\ d\ PP\ R;$
$(V,V') \in R;\ i < length\ V\ ]\!]$
$\Longrightarrow (NDC\ d\ (V!i) \vee IDC\ d\ (V!i)\ (htchLoc\ (pp\ (V!i))))$
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)

lemma *WHATWHERE-empty*:
WHATWHERE-Secure []
by (*simp add*: *WHATWHERE-Secure-def*, *auto*,
rule-tac x={([],[])} **in** *exI*,
simp add: *Strong-dlHPP-Bisimulation-def sym-def trans-def*)

end

end

2.2 Proof technique for compositionality results

For proving compositionality results for WHAT&WHERE-security, we formalize the following "up-to technique" and prove it sound:

theory *Up-To-Technique*
imports *WHATWHERE-Security*
begin

context *WHATWHERE*
begin

abbreviation *SdlHPPB* **where** *SdlHPPB* $\equiv$ *Strong-dlHPP-Bisimulation*

— define the 'reflexive part' of a relation (sets of elements which are related with themselves by the given relation)
definition *Arefl* :: $('a \times 'a)\ set \Rightarrow 'a\ set$
where
$Arefl\ R = \{e.\ (e,e) \in R\}$

lemma *commonArefl-subset-commonDomain*:
$(Arefl\ R1 \cap Arefl\ R2) \subseteq (Domain\ R1 \cap Domain\ R2)$
by (*simp add*: *Arefl-def*, *auto*)

— define disjoint strong (d,lH,PP)-bisimulation up-to-R' for a relation R
definition *disj-dlHPP-Bisimulation-Up-To-R'* ::
$'d \Rightarrow nat\ set \Rightarrow 'com\ Bisimulation\text{-}type$

$\Rightarrow$ *'com Bisimulation-type* $\Rightarrow$ *bool*
where
disj-dlHPP-Bisimulation-Up-To-R′ d PP R′ R $\equiv$
SdlHPPB d PP R′ $\wedge$ (*sym R*) $\wedge$ (*trans R*)
$\wedge$ $(\forall (V,V') \in R.\ length\ V = length\ V') \wedge$
$(\forall (V,V') \in R.\ \forall i < length\ V.$
$((NDC\ d\ (V!i)) \vee$
$(IDC\ d\ (V!i)\ (htchLoc\ (pp\ (V!i)))))) \wedge$
$(\forall (V,V') \in R.\ \forall i < length\ V.\ \forall m1\ m1'\ m2\ \alpha\ p.$
$((\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle \wedge m1 \sim_{d,(htchLocSet\ PP)} m1')$
$\longrightarrow (\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$(stepResultsinR\ p\ p'\ (R \cup R')) \wedge (\alpha,\alpha') \in (R \cup R') \wedge$
$(dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2')))$

— lemma about the transitivity of the union of symmetric and transitive relations under certain circumstances
lemma *trans-RuR′*:
assumes *transR*: *trans R*
assumes *symR*: *sym R*
assumes *transR′*: *trans R′*
assumes *symR′*: *sym R′*
assumes *eqlenR*: $\forall (V,V') \in R.\ length\ V = length\ V'$
assumes *eqlenR′*: $\forall (V,V') \in R'.\ length\ V = length\ V'$
assumes *Areflassump*: $(Arefl\ R \cap Arefl\ R') \subseteq \{[]\}$
shows *trans* $(R \cup R')$
proof −
{
fix $V\ V'\ V''$
assume *p1*: $(V,V') \in (R \cup R')$
assume *p2*: $(V',V'') \in (R \cup R')$

from *p1 p2* **have** $(V,V'') \in (R \cup R')$
proof (*auto*)
assume *inR1*: $(V,V') \in R$
assume *inR2*: $(V',V'') \in R$
from *inR1 inR2 transR* **show** $(V,V'') \in R$
unfolding *trans-def*
by *blast*
next
assume *inR′1*: $(V,V') \in R'$
assume *inR′2*: $(V',V'') \in R'$
assume *notinR′*: $(V,V'') \notin R'$
from *inR′1 inR′2 transR′* **have** *inR′*: $(V,V'') \in R'$
unfolding *trans-def*
by *blast*
from *notinR′ inR′* **have** *False*
by *auto*
thus $(V,V'') \in R$ **..**
next

assume *inR1*: $(V,V') \in R$
assume *inR'2*: $(V',V'') \in R'$
from *inR1 symR transR* **have** $(V,V) \in R \wedge (V',V') \in R$
unfolding *sym-def trans-def*
by *blast*
hence *AreflR*: $\{V,V'\} \subseteq$ *Arefl* R **by** (*simp add*: *Arefl-def*)
from *inR'2 symR' transR'* **have** $(V',V') \in R' \wedge (V'',V'') \in R'$
unfolding *sym-def trans-def*
by *blast*
hence *AreflR'*: $\{V',V''\} \subseteq$ *Arefl* R' **by** (*simp add*: *Arefl-def*)

from *AreflR AreflR' Areflassump* **have** *V'empt*: $V' = []$
by (*simp add*: *Arefl-def*, *blast*)
with *inR'2 eqlenR'* **have** $V' = V''$ **by** *auto*
with *inR1* **show** $(V,\ V'') \in R$ **by** *auto*
next
assume *inR'1*: $(V,V') \in R'$
assume *inR2*: $(V',V'') \in R$
from *inR'1 symR' transR'* **have** $(V,V) \in R' \wedge (V',V') \in R'$
unfolding *sym-def trans-def*
by *blast*
hence *AreflR'*: $\{V,V'\} \subseteq$ *Arefl* R' **by** (*simp add*: *Arefl-def*)
from *inR2 symR transR* **have** $(V',V') \in R \wedge (V'',V'') \in R$
unfolding *sym-def trans-def*
by *blast*
hence *AreflR*: $\{V',V''\} \subseteq$ *Arefl* R **by** (*simp add*: *Arefl-def*)

from *AreflR AreflR' Areflassump* **have** *V'empt*: $V' = []$
by (*simp add*: *Arefl-def*, *blast*)
with *inR'1 eqlenR'* **have** $V' = V$ **by** *auto*
with *inR2* **show** $(V,\ V'') \in R$ **by** *auto*
qed
}
thus *?thesis* **unfolding** *trans-def* **by** *force*

qed

lemma *Up-To-Technique*:
$[\![$ *disj-dlHPP-Bisimulation-Up-To-R' d PP R' R*;
Arefl $R \cap$ *Arefl* $R' \subseteq \{[]\}$ $]\!]$
$\Longrightarrow$ *SdlHPPB d PP* $(R \cup R')$
proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-R'-def*
Strong-dlHPP-Bisimulation-def, *auto*)
assume *symR'*: *sym* R'
assume *symR*: *sym* R
with *symR'* **show** *sym* $(R \cup R')$
by (*simp add*: *sym-def*)
next
assume *symR'*: *sym* R'

assume *symR*: *sym R*
assume *transR'*: *trans R'*
assume *transR*: *trans R*
assume *eqlenR'*: $\forall (V,\ V')\in R'.$ *length V = length V'*
assume *eqlenR*: $\forall (V,\ V')\in R.$ *length V = length V'*
assume *areflassump*: *Arefl R* $\cap$ *Arefl R'* $\subseteq$ {[]}
from *symR' symR transR' transR eqlenR' eqlenR areflassump trans-RuR'*
show *trans* $(R \cup R')$
by *blast*
— condition about IDC and NDC and equal length already proven above by auto tactic!
next
fix *V V' i m1 m1' m2* α *p*
assume *inR*: $(V,V') \in R$
assume *irange*: $i <$ *length V*
assume *step*: $\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,(htchLocSet\ PP)} m1'$
assume *disjBisimUpTo*: $\forall (V,V')\in R.\ \forall i <$ *length V*. $\forall$ *m1 m1' m2* α *p*.
$\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle \wedge m1 \sim_{d,(htchLocSet\ PP)} m1' \longrightarrow$
$(\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
stepResultsinR p p' $(R \cup R') \wedge ((\alpha,\ \alpha') \in R \vee (\alpha,\ \alpha') \in R') \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2')
from *inR irange step dhequal disjBisimUpTo* **show** $\exists p'\ \alpha'\ m2'.$
$\langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$ *stepResultsinR p p'* $(R \cup R') \wedge$
$((\alpha,\alpha') \in R \vee (\alpha,\alpha') \in R') \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by *blast*
next
fix *V V' i m1 m1' m2* α *p*
assume *inR'*: $(V,V') \in R'$
assume *irange*: $i <$ *length V*
assume *step*: $\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,(htchLocSet\ PP)} m1'$
assume *disjBisimUpTo*: $\forall (V,V')\in R'.\ \forall i <$ *length V*. $\forall$ *m1 m1' m2* α *p*.
$\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle \wedge m1 \sim_{d,(htchLocSet\ PP)} m1' \longrightarrow$
$(\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
stepResultsinR p p' R' $\wedge (\alpha,\ \alpha') \in R' \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2')
from *inR' irange step dhequal disjBisimUpTo* **show** $\exists p'\ \alpha'\ m2'.$
$\langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$ *stepResultsinR p p'* $(R \cup R') \wedge$
$((\alpha,\alpha') \in R \vee (\alpha,\alpha') \in R') \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add*: *stepResultsinR-def*,*fastforce*)
qed

lemma *Union-Strong-dlHPP-Bisim*:
⟦ *SdlHPPB d PP R*; *SdlHPPB d PP R'*;
Arefl R $\cap$ *Arefl R'* $\subseteq$ {[]} ⟧

$\Longrightarrow$ *SdlHPPB d PP* $(R \cup R')$
by (*rule Up-To-Technique*, *simp-all add*:
disj-dlHPP-Bisimulation-Up-To-R′-def
Strong-dlHPP-Bisimulation-def stepResultsinR-def, *fastforce*)

lemma *adding-emptypair-keeps-SdlHPPB*:
assumes *SdlHPP*: *SdlHPPB d PP R*
shows *SdlHPPB d PP* $(R \cup \{([],[])\})$
proof −
have *SdlHPPemp*: *SdlHPPB d PP* $\{([],[])\}$
by (*simp add*: *Strong-dlHPP-Bisimulation-def sym-def trans-def*)

have *commonDom*: *Domain* $R \cap$ *Domain* $\{([],[])\} \subseteq \{[]\}$
by *auto*

with *commonArefl-subset-commonDomain* **have** *Areflassump*:
Arefl $R \cap$ *Arefl* $\{([],[])\} \subseteq \{[]\}$
by *force*

with *SdlHPP SdlHPPemp Union-Strong-dlHPP-Bisim* **show**
SdlHPPB d PP $(R \cup \{([],[])\})$
by *force*
qed

end

end

2.3 Proof of parallel compositionality

We prove that WHAT&WHERE-security is preserved under composition of WHAT&WHERE-secure threads.

theory *Parallel-Composition*
imports *Up-To-Technique MWLs*
begin

locale *WHATWHERE-Secure-Programs* =
L : *MWLs-semantics E BMap*
\+ *WWs* : *WHATWHERE MWLsSteps-det E pp DA lH*
for *E* :: (*′exp*, *′id*, *′val*) *Evalfunction*
and *BMap* :: *′val* ⇒ *bool*
and *DA* :: (*′id*, *′d*::*order*) *DomainAssignment*
and *lH* :: (*′d*, *′exp*) *lHatches*
begin

lemma *SdlHPPB-restricted-on-PP-is-SdlHPPB*:

assumes *SdlHPPB*: *SdlHPPB d PP R′*
assumes *inR′*: $(V,V) \in R'$
assumes *Rdef*: $R = \{(V',V''). (V',V'') \in R'$
$\wedge$ *set* $(PPV\ V') \subseteq$ *set* $(PPV\ V)$
$\wedge$ *set* $(PPV\ V'') \subseteq$ *set* $(PPV\ V)\}$
shows *SdlHPPB d PP R*
proof (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
from *SdlHPPB* **have** *sym R′*
by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
with *Rdef* **show** *sym R*
by (*simp add*: *sym-def*)
next
from *SdlHPPB* **have** *trans R′*
by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
with *Rdef* **show** *trans R*
by (*simp add*: *trans-def*, *auto*)
next
fix $V'\ V''$
assume *inR-part*: $(V',V'') \in R$
with *SdlHPPB Rdef* **show** *length* V' = *length* V''
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
next
fix $V'\ V''\ i$
assume *inR-part*: $(V',V'') \in R$
assume *irange*: $i <$ *length* V'
assume *notIDC*:
$\neg$ *IDC d* $(V'!i)$ (*htchLoc* (*pp* $(V'!i)$))
with *SdlHPPB inR-part irange Rdef*
show *NDC d* $(V'!i)$
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
next
fix $V'\ V''\ i\ \alpha\ p\ m1\ m1'\ m2$
assume *inR-part*: $(V',V'') \in R$
assume *irange*: $i <$ *length* V'
assume *step*: $\langle V'!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,(htchLocSet\ PP)} m1'$

from *inR-part SdlHPPB Rdef* **have** *eqlen*: *length* V' = *length* V''
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)

from *inR-part Rdef*
have *set* $(PPV\ V') \subseteq$ *set* $(PPV\ V) \wedge$ *set* $(PPV\ V'') \subseteq$ *set* $(PPV\ V)$
by *auto*

with *irange PPc-in-PPV-version eqlen*
have *PPc-Vs-at-i*:
set $(PPc\ (V'!i)) \subseteq$ *set* $(PPV\ V) \wedge$ *set* $(PPc\ (V''!i)) \subseteq$ *set* $(PPV\ V)$
by (*metis subset-trans*)

from *SdlHPPB inR-part Rdef irange step dhequal*
strongdlHPPB-aux[*of d PP R′ i*
V′ V′′ m1 α p m2 m1′]
obtain *p′ α′ m2′* **where** *stepreq*: ⟨*V′′*!*i*,*m1′*⟩ →⊲*α′*⊳ ⟨*p′*,*m2′*⟩ ∧
stepResultsinR p p′ R′ ∧ (*α*,*α′*) ∈ *R′* ∧
dhequality-alternative d PP (*pp* (*V′*!*i*)) *m2 m2′*
by *auto*
have *Rpp′*: *stepResultsinR p p′ R*
proof −
{
fix *c c′*
assume *step1*: ⟨*V′*!*i*,*m1*⟩ →⊲*α*⊳ ⟨*Some c*,*m2*⟩
assume *step2*: ⟨*V′′*!*i*,*m1′*⟩ →⊲*α′*⊳ ⟨*Some c′*,*m2′*⟩
assume *inR′-res*: ([*c*],[*c′*]) ∈ *R′*

from *PPc-Vs-at-i step1 step2 PPsc-of-step*
have *set* (*PPc c*) ⊆ *set* (*PPV V*) ∧ *set* (*PPc c′*) ⊆ *set* (*PPV V*)
by (*metis* (*no-types*) *the.simps xt1*(*6*))

with *inR′-res Rdef* **have** ([*c*],[*c′*]) ∈ *R*
by *auto*
}
thus *?thesis*
by (*metis step stepResultsinR-def stepreq*)
qed

have *Rαα′*: (*α*,*α′*) ∈ *R*
proof −
from *PPc-Vs-at-i step stepreq PPsα-of-step* **have**
set (*PPV α*) ⊆ *set* (*PPV V*) ∧ *set* (*PPV α′*) ⊆ *set* (*PPV V*)
by (*metis* (*no-types*) *xt1*(*6*))
with *stepreq Rdef* **show** *?thesis*
by *auto*
qed

from *stepreq Rpp′ Rαα′* **show**
∃ *p′ α′ m2′*. ⟨*V′′*!*i*,*m1′*⟩ →⊲*α′*⊳ ⟨*p′*,*m2′*⟩ ∧
stepResultsinR p p′ R ∧ (*α*,*α′*) ∈ *R* ∧
dhequality-alternative d PP (*pp* (*V′*!*i*)) *m2 m2′*
by *auto*
qed

theorem *parallel-composition*:
⟦ ∀ *i* < *length V*. *WHATWHERE-Secure* [*V*!*i*]; *unique-PPV V* ⟧
⟹ *WHATWHERE-Secure V*
proof (*simp add*: *WHATWHERE-Secure-def*, *induct V*, *auto*)
fix *d PP*
from *WHATWHERE-empty*

show $\exists R.$ *SdlHPPB d PP R* $\wedge$ ([],[]) $\in R$
by (*simp add*: *WHATWHERE-Secure-def*)
next
fix *c V d PP*
assume *IH*: $[\![\ \forall i < length\ V.$
$\forall d\ PP.\ \exists R.\ SdlHPPB\ d\ PP\ R \wedge ([V!i],[V!i]) \in R;$
unique-PPV V $]\!]$
$\Longrightarrow \forall d\ PP.\ \exists R.\ SdlHPPB\ d\ PP\ R \wedge (V,V) \in R$
assume *ISassump*: $\forall i < Suc\ (length\ V).$
$\forall d\ PP.\ \exists R.\ SdlHPPB\ d\ PP\ R \wedge ([(c\#V)!i],[(c\#V)!i]) \in R$
assume *uniPPcV*: *unique-PPV* $(c\#V)$

hence *IHassump1*: *unique-PPV V*
by (*simp add*: *unique-PPV-def*)

from *uniPPcV* **have** *nocommonPP*: *set* (*PPc c*) $\cap$ *set* (*PPV V*) = {}
by (*simp add*: *unique-PPV-def*)

from *ISassump* **have** *IHassump2*: $\forall i < length\ V.$
$\forall d\ PP.\ \exists R.\ SdlHPPB\ d\ PP\ R \wedge ([V!i],[V!i]) \in R$
by *auto*

with *IHassump1 IH* **obtain** *RV'* **where** *RV'assump*:
SdlHPPB d PP RV' $\wedge$ $(V,V) \in RV'$
by *blast*

def $RV \equiv \{(V',V'').\ (V',V'') \in RV' \wedge set\ (PPV\ V') \subseteq set\ (PPV\ V)$
$\wedge\ set\ (PPV\ V'') \subseteq set\ (PPV\ V)\}$

with *RV'assump RV-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPRV*: *SdlHPPB d PP RV*
by *force*

from *ISassump* **obtain** *Rc'* **where** *Rc'assump*:
SdlHPPB d PP Rc' $\wedge$ ([c],[c]) $\in$ *Rc'*
by (*metis append-Nil drop-Nil neq0-conv not-Cons-self*
nth-append-length nth-drop' zero-less-Suc)

def $Rc \equiv \{(V',V'').\ (V',V'') \in Rc' \wedge set\ (PPV\ V') \subseteq set\ (PPc\ c)$
$\wedge\ set\ (PPV\ V'') \subseteq set\ (PPc\ c)\}$

with *Rc'assump Rc-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPRc*: *SdlHPPB d PP Rc*
by *force*

from *nocommonPP* **have** *Domain RV* $\cap$ *Domain Rc* $\subseteq$ {[]}
by (*simp add*: *RV-def Rc-def*, *auto*,
metis Int-absorb2 Int-mono inf-commute inf-idem le-bot nocommonPP
order-eq-refl unique-V-uneq)

with *commonArefl-subset-commonDomain*
have *Areflassump1*: *Arefl RV* ∩ *Arefl Rc* ⊆ {[]}
  by *force*

def $R \equiv \{(V',V''). \exists\, c\ c'\ W\ W'.\ V' = c\#W \wedge V'' = c'\#W' \wedge W \neq []$
  $\wedge\ W' \neq [] \wedge ([c],[c']) \in Rc \wedge (W,W') \in RV\}$

with *RV-def RV′assump Rc-def Rc′assump* **have** *inR*:
  $V \neq [] \Longrightarrow (c\#V,c\#V) \in R$
  by *auto*

from *R-def Rc-def RV-def nocommonPP*
have *Domain R* ∩ *Domain* (*Rc* ∪ *RV*) = {}
  by (*simp add*: *R-def Rc-def RV-def*, *auto*,
    metis inf-bot-right le-inf-iff subset-empty unique-V-uneq,
    metis (*hide-lams*, *no-types*) *inf-absorb1 inf-bot-right le-inf-iff unique-c-uneq*)

with *commonArefl-subset-commonDomain*
have *Areflassump2*: *Arefl R* ∩ *Arefl* (*Rc* ∪ *RV*) ⊆ {[]}
  by *force*

have *disjuptoR*:
  disj-dlHPP-Bisimulation-Up-To-R′ d PP (*Rc* ∪ *RV*) *R*
  proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-R′-def*, *auto*)
      from *Areflassump1 SdlHPPRc SdlHPPRV Union-Strong-dlHPP-Bisim*
      show *SdlHPPB d PP* (*Rc* ∪ *RV*)
        by *force*
    next
      from *SdlHPPRV* **have** *symRV*: *sym RV*
        by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
      from *SdlHPPRc* **have** *symRc*: *sym Rc*
        by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
      with *symRV R-def* **show** *sym R*
        by (*simp add*: *sym-def*, *auto*)
    next
      from *SdlHPPRV* **have** *transRV*: *trans RV*
        by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
      from *SdlHPPRc* **have** *transRc*: *trans Rc*
        by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
      show *trans R*
        proof −
        {
        fix $V\ V'\ V''$
        assume *p1*: $(V,V') \in R$
        assume *p2*: $(V',V'') \in R$
        have $(V,V'') \in R$
          proof −
            from *p1 R-def* **obtain** $c\ c'\ W\ W'$ **where** *p1assump*:

$V = c\# W \wedge V' = c'\# W' \wedge W \neq [] \wedge W' \neq [] \wedge$
$([c],[c']) \in Rc \wedge (W,W') \in RV$
by *auto*
with *p2 R-def* **obtain** c'' W'' **where** *p2assump*:
$V'' = c''\# W'' \wedge W'' \neq [] \wedge$
$([c'],[c'']) \in Rc \wedge (W',W'') \in RV$
by *auto*
with *p1assump transRc transRV* **have**
trans-assump: $([c],[c'']) \in Rc \wedge (W,W'') \in RV$
by (*simp add*: *trans-def*, *blast*)
with *p1assump p2assump R-def* **show** *?thesis*
by *auto*
qed
}
thus *?thesis* **unfolding** *trans-def* **by** *blast*
qed
next
fix V V'
assume $(V,V') \in R$
with *R-def SdlHPPRV* **show** *length* V = *length* V'
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
next
fix V V' i
assume *inR*: $(V,V') \in R$
assume *irange*: $i <$ *length* V
assume *notIDC*: $\neg$ *IDC d* $(V!i)$
(*htchLoc* (*pp* $(V!i)$))
from *inR R-def* **obtain** c c' W W' **where** *VV'assump*:
$V = c\# W \wedge V'=c'\# W' \wedge W \neq [] \wedge W' \neq [] \wedge$
$([c],[c']) \in Rc \wedge (W,W') \in RV$
by *auto*
— Case separation for i
from *VV'assump SdlHPPRc* **have** *Case-i0*:
$i = 0 \Longrightarrow$ (*NDC d* $(V!i) \vee$
IDC d $(V!i)$ (*htchLoc* (*pp* $(V!i)$)))
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)

from *VV'assump SdlHPPRV* **have** $\forall\, i <$ *length* W.
(*NDC d* $(W!i) \vee$
IDC d $(W!i)$ (*htchLoc* (*pp* $(W!i)$)))
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)

with *irange VV'assump* **have** *Case-in0*:
$i > 0 \Longrightarrow$ (*NDC d* $(V!i) \vee$
IDC d $(V!i)$ (*htchLoc* (*pp* $(V!i)$)))
by *simp*
from *notIDC Case-i0 Case-in0*
show *NDC d* $(V!i)$
by *auto*

next
fix $V\ V'\ m1\ m1'\ m2\ \alpha\ p\ i$
assume *inR*: $(V,V') \in R$
assume *irange*: $i < length\ V$
assume *step*: $\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,(htchLocSet\ PP)} m1'$

from *inR R-def* **obtain** $c\ c'\ W\ W'$ **where** *VV'assump*:
$V = c\#W \wedge V'=c'\#W' \wedge W \neq [] \wedge W' \neq [] \wedge$
$([c],[c']) \in Rc \wedge (W,W') \in RV$
by *auto*
— Case separation for i
from *VV'assump SdlHPPRc strongdlHPPB-aux*[*of d PP Rc 0* [*c*] [*c'*]] *step dhequal*
have *Case-i0*:
$i = 0 \Longrightarrow \exists\, p'\ \alpha'\ m2'.$
$\langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R \cup (Rc \cup RV)) \wedge$
$((\alpha,\alpha') \in R \vee (\alpha,\alpha') \in Rc \vee (\alpha,\alpha') \in RV) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
by (*simp add*: *stepResultsinR-def*, *blast*)

from *step VV'assump irange* **have** *rewV*:
$i > 0 \Longrightarrow (i-Suc\ 0) < length\ W \wedge V!i = W!(i-Suc\ 0)$
by *simp*

with *irange VV'assump step dhequal SdlHPPRV strongdlHPPB-aux*[*of d PP RV - W W'*]
have *Case-in0*:
$i > 0 \Longrightarrow \exists\, p'\ \alpha'\ m2'.$
$\langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R \cup (Rc \cup RV)) \wedge$
$((\alpha,\alpha') \in R \vee (\alpha,\alpha') \in Rc \vee (\alpha,\alpha') \in RV) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
by (*simp add*: *stepResultsinR-def*, *blast*)

from *Case-i0 Case-in0*
show $\exists\, p'\ \alpha'\ m2'.$
$\langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R \cup (Rc \cup RV)) \wedge$
$((\alpha,\alpha') \in R \vee (\alpha,\alpha') \in Rc \vee (\alpha,\alpha') \in RV) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
by *auto*
qed
with *Areflassump2 Rc'assump Up-To-Technique*
show $\exists\, R.\ SdlHPPB\ d\ PP\ R \wedge (c\#V,\ c\#V) \in R$
by (*metis UnCI inR*)

qed

end

end

3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

As in [2], we instantiate the language with a simple while language that supports dynamic thread creation via a spawn command (Multi-threaded While Language with spawn, MWLs). Note that the language is still parametric in the language used for Boolean and arithmetic expressions (*'exp*).

```
theory MWLs
imports ../Strong−Security/Types
begin

— type parameters not instantiated:
— 'exp: expressions (arithmetic, boolean...)
— 'val: numbers, boolean constants....
— 'id: identifier names

— SYNTAX

datatype ('exp, 'id) MWLsCom
  = Skip nat (skip_ [50] 70)
  | Assign 'id nat 'exp
      (_:=_ _ [70,50,70] 70)

  | Seq ('exp, 'id) MWLsCom
        ('exp, 'id) MWLsCom
      (_;_ [61,60] 60)

  | If-Else nat 'exp ('exp, 'id) MWLsCom
        ('exp, 'id) MWLsCom
      (if_ _ then _ else _ fi [50,80,79,79] 70)

  | While-Do nat 'exp ('exp, 'id) MWLsCom
      (while_ _ do _ od [50,80,79] 70)

  | Spawn nat (('exp, 'id) MWLsCom) list
      (spawn_ _ [50,70] 70)

— function for obtaining the program point of some MWLsloc command
primrec pp ::('exp, 'id) MWLsCom ⇒ nat
where
```

$pp\ (skip_\iota) = \iota\ |$
$pp\ (x :=_\iota e) = \iota\ |$
$pp\ (c1;c2) = pp\ c1\ |$
$pp\ (if_\iota\ b\ then\ c1\ else\ c2\ fi) = \iota\ |$
$pp\ (while_\iota\ b\ do\ c\ od) = \iota\ |$
$pp\ (spawn_\iota\ V) = \iota$

— mutually recursive functions to collect program points of commands and thread pools
primrec $PPc :: ('exp,'id)\ MWLsCom \Rightarrow nat\ list$
and $PPV :: ('exp,'id)\ MWLsCom\ list \Rightarrow nat\ list$
where
$PPc\ (skip_\iota) = [\iota]\ |$
$PPc\ (x :=_\iota e) = [\iota]\ |$
$PPc\ (c1;c2) = (PPc\ c1)\ @\ (PPc\ c2)\ |$
$PPc\ (if_\iota\ b\ then\ c1\ else\ c2\ fi) = [\iota]\ @\ (PPc\ c1)\ @\ (PPc\ c2)\ |$
$PPc\ (while_\iota\ b\ do\ c\ od) = [\iota]\ @\ (PPc\ c)\ |$
$PPc\ (spawn_\iota\ V) = [\iota]\ @\ (PPV\ V)\ |$

$PPV\ [] = []\ |$
$PPV\ (c\#V) = (PPc\ c)\ @\ (PPV\ V)$

— predicate indicating that a command only contains unique program points
definition $unique\text{-}PPc :: ('exp, 'id)\ MWLsCom \Rightarrow bool$
where
$unique\text{-}PPc\ c = distinct\ (PPc\ c)$

— predicate indicating that a thread pool only contains unique program points
definition $unique\text{-}PPV :: ('exp, 'id)\ MWLsCom\ list \Rightarrow bool$
where
$unique\text{-}PPV\ V = distinct\ (PPV\ V)$

lemma $PPc\text{-}nonempt$: $PPc\ c \neq []$
  by $(induct, simp\text{-}all)$

lemma $unique\text{-}c\text{-}uneq$: $set\ (PPc\ c) \cap set\ (PPc\ c') = \{\} \Longrightarrow c \neq c'$
  by $(insert\ PPc\text{-}nonempt, force)$

lemma $V\text{-}nonempt\text{-}PPV\text{-}nonempt$: $V \neq [] \Longrightarrow PPV\ V \neq []$
  by $(auto, induct\ V, simp\text{-}all, insert\ PPc\text{-}nonempt, force)$

lemma $unique\text{-}V\text{-}uneq$:
$[\![V \neq []; V' \neq []; set\ (PPV\ V) \cap set\ (PPV\ V') = \{\}]\!] \Longrightarrow V \neq V'$
  by $(auto, induct\ V, simp\text{-}all, insert\ V\text{-}nonempt\text{-}PPV\text{-}nonempt, auto)$

lemma $PPc\text{-}in\text{-}PPV$: $c \in set\ V \Longrightarrow set\ (PPc\ c) \subseteq set\ (PPV\ V)$
  by $(induct\ V, auto)$

lemma $listindices\text{-}aux$: $i < length\ V \Longrightarrow (V!i) \in set\ V$

by (*metis nth-mem*)

lemma *PPc-in-PPV-version*:
$i < length\ V \Longrightarrow set\ (PPc\ (V!i)) \subseteq set\ (PPV\ V)$
by (*rule PPc-in-PPV*, *erule listindices-aux*)

lemma *uniPPV-uniPPc*: $unique\text{-}PPV\ V \Longrightarrow (\forall\, i < length\ V.\ unique\text{-}PPc\ (V!i))$
by (*auto*, *simp add*: *unique-PPV-def*, *induct V*,
auto simp add: *unique-PPc-def*,
metis in-set-conv-nth length-Suc-conv set-ConsD)

— SEMANTICS

locale *MWLs-semantics* =
fixes E :: (*'exp*, *'id*, *'val*) *Evalfunction*
and *BMap* :: *'val* ⇒ *bool*
begin

— steps semantics, set of deterministic steps from commands to program states
inductive-set
MWLsSteps-det ::
(*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLsCom*) *TLSteps*
and *MWLslocSteps-det'* ::
(*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLsCom*) *TLSteps-curry*
((*1*⟨-,/-⟩) →◁-▷/ (*1*⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
$\langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle c2,m2\rangle \equiv ((c1,m1),\alpha,(c2,m2)) \in MWLsSteps\text{-}det$ |
skip: $\langle skip_\iota,m\rangle \rightarrow\triangleleft[]\triangleright \langle None,m\rangle$ |
assign: $(E\ e\ m) = v \Longrightarrow$
$\langle x :=_\iota e,m\rangle \rightarrow\triangleleft[]\triangleright \langle None,m(x := v)\rangle$ |
seq1: $\langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle None,m'\rangle \Longrightarrow$
$\langle c1;c2,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ c2,m'\rangle$ |
seq2: $\langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ c1',m'\rangle \Longrightarrow$
$\langle c1;c2,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ (c1';c2),m'\rangle$ |
iftrue: $BMap\ (E\ b\ m) = True \Longrightarrow$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\triangleleft[]\triangleright \langle Some\ c1,m\rangle$ |
iffalse: $BMap\ (E\ b\ m) = False \Longrightarrow$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\triangleleft[]\triangleright \langle Some\ c2,m\rangle$ |
whiletrue: $BMap\ (E\ b\ m) = True \Longrightarrow$
$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\triangleleft[]\triangleright \langle Some\ (c;(while_\iota\ b\ do\ c\ od)),m\rangle$ |
whilefalse: $BMap\ (E\ b\ m) = False \Longrightarrow$
$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\triangleleft[]\triangleright \langle None,m\rangle$ |
spawn: $\langle spawn_\iota\ V,m\rangle \rightarrow\triangleleft V\triangleright \langle None,m\rangle$

inductive-cases *MWLsSteps-det-cases*:
$\langle skip_\iota,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle x :=_\iota e,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle c1;c2,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\langle if_\iota\ b\ then\ c1\ else\ c2\ fi,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$

$\langle while_\iota\ b\ do\ c\ od,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m'\rangle$
$\langle spawn_\iota\ V,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m'\rangle$

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)
inductive-set
MWLsSteps-ndet ::
  ('*exp*, '*id*, '*val*, ('*exp*, '*id*) *MWLsCom*) *TPSteps*
and *MWLsSteps-ndet'* ::
  ('*exp*, '*id*, '*val*, ('*exp*, '*id*) *MWLsCom*) *TPSteps-curry*
$((1\langle\text{-},/\text{-}\rangle) \Rightarrow/\ (1\langle\text{-},/\text{-}\rangle)\ [0,0,0,0]\ 81)$
where
$\langle V,m\rangle \Rightarrow \langle V',m'\rangle \equiv ((V,m),(V',m')) \in$ *MWLsSteps-ndet* |
stepthreadi1: $\langle ci,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle None,m'\rangle \Longrightarrow$
  $\langle cf\ @\ [ci]\ @\ ca,m\rangle \Rightarrow \langle cf\ @\ \alpha\ @\ ca,m'\rangle$ |
stepthreadi2: $\langle ci,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle Some\ c',m'\rangle \Longrightarrow$
  $\langle cf\ @\ [ci]\ @\ ca,m\rangle \Rightarrow \langle cf\ @\ [c']\ @\ \alpha\ @\ ca,m\rangle$

— lemma about existence and uniqueness of next memory of a step
lemma *nextmem-exists-and-unique*:
$\exists\, m'\ p\ \alpha.\ \langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m'\rangle$
  $\wedge\ (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m''\rangle) \longrightarrow m'' = m')$
  by (*induct*, *auto*, *metis MWLsSteps-det.skip MWLsSteps-det-cases*(*1*),
    metis MWLsSteps-det-cases(*2*) *MWLsSteps-det.assign*,
    metis (*no-types*) *MWLsSteps-det.seq1 MWLsSteps-det.seq2*
    MWLsSteps-det-cases(*3*) *not-Some-eq*,
    metis MWLsSteps-det.iffalse MWLsSteps-det.iftrue
    MWLsSteps-det-cases(*4*),
    metis MWLsSteps-det.whilefalse MWLsSteps-det.whiletrue
    MWLsSteps-det-cases(*5*),
    metis MWLsSteps-det.spawn MWLsSteps-det-cases(*6*))

lemma *PPsc-of-step*:
$[\![\ \langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m'\rangle;\ \exists\, c'.\ p = Some\ c'\ ]\!]$
  $\Longrightarrow set\ (PPc\ (the\ p)) \subseteq set\ (PPc\ c)$
  by (*induct rule*: *MWLsSteps-det.induct*, *auto*)

lemma *PPsα-of-step*:
$\langle c,m\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m'\rangle$
  $\Longrightarrow set\ (PPV\ \alpha) \subseteq set\ (PPc\ c)$
  by (*induct rule*: *MWLsSteps-det.induct*, *auto*)

end

end

3.2 Proofs of atomic compositionality results

We prove for each atomic command of our example programming language (i.e. a command that is not composed out of other commands) that it is strongly secure if the expressions involved are indistinguishable for an observer on security level d.

theory *WHATWHERE-Secure-Skip-Assign*
imports *Parallel-Composition*
begin

context *WHATWHERE-Secure-Programs*
begin

abbreviation *NextMem'*
$(\llbracket\text{-}\rrbracket'(\text{-}'))$
where
$\llbracket c \rrbracket(m) \equiv \mathit{NextMem}\ c\ m$

— define when two expressions are indistinguishable with respect to a domain d
definition *d-indistinguishable* :: $'d::order \Rightarrow {}'exp \Rightarrow {}'exp \Rightarrow bool$
where
$\mathit{d\text{-}indistinguishable}\ d\ e1\ e2 \equiv \forall\, m\ m'.\ ((m =_d m')$
$\longrightarrow ((E\ e1\ m) = (E\ e2\ m')))$

abbreviation *d-indistinguishable'* :: $'exp \Rightarrow {}'d::order \Rightarrow {}'exp \Rightarrow bool$
$(\ (\text{-} \equiv_{\text{-}} \text{-})\ )$
where
$e1 \equiv_d e2 \equiv \mathit{d\text{-}indistinguishable}\ d\ e1\ e2$

— symmetry of d-indistinguishable
lemma *d-indistinguishable-sym*:
$e \equiv_d e' \Longrightarrow e' \equiv_d e$
by (*simp add*: *d-indistinguishable-def d-equal-def*, *metis*)

— transitivity of d-indistinguishable
lemma *d-indistinguishable-trans*:
$\llbracket\ e \equiv_d e';\ e' \equiv_d e''\ \rrbracket \Longrightarrow e \equiv_d e''$
by (*simp add*: *d-indistinguishable-def d-equal-def*, *metis*)

— predicate for dH-indistinguishable
definition *dH-indistinguishable* :: $'d \Rightarrow ('d,\ 'exp)\ \mathit{Hatches}$
$\Rightarrow {}'exp \Rightarrow {}'exp \Rightarrow bool$
where
$\mathit{dH\text{-}indistinguishable}\ d\ H\ e1\ e2 \equiv (\forall\, m\ m'.\ m \sim_{d,H} m'$
$\longrightarrow (E\ e1\ m = E\ e2\ m'))$

abbreviation *dH-indistinguishable'* :: $'exp \Rightarrow {}'d$

$\Rightarrow$ ('d, 'exp) *Hatches* $\Rightarrow$ 'exp $\Rightarrow$ *bool*
((- $\equiv_{-,-}$ -))
where
e1 $\equiv_{d,H}$ *e2* $\equiv$ *dH-indistinguishable d H e1 e2*

lemma *empH-implies-dHindistinguishable-eq-dindistinguishable*:
($e \equiv_{d,\{\}} e'$) = ($e \equiv_{d} e'$)
by (*simp add*: *d-indistinguishable-def dH-indistinguishable-def*
dH-equal-def d-equal-def)

theorem *WHATWHERE-Secure-Skip*:
WHATWHERE-Secure [$skip_\iota$]
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
fix *d PP*
def *R* $\equiv$ {(*V*::('exp,'id) *MWLsCom list*,*V'*::('exp,'id) *MWLsCom list*).
V = *V'* $\wedge$ (*V* = [] $\vee$ *V* = [$skip_\iota$])}

have *inR*: ([$skip_\iota$],[$skip_\iota$]) $\in$ *R*
by (*simp add*: *R-def*)

have *SdlHPPB d PP R*
proof (*simp add*: *Strong-dlHPP-Bisimulation-def R-def sym-def trans-def*
NDC-def NextMem-def, *auto*)
fix *m1 m1'*
assume *dequal*: *m1* $=_d$ *m1'*
have *nextm1*: (*THE m2*. ($\exists$ *p* α. $\langle skip_\iota$,*m1*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p*,*m2*$\rangle$)) = *m1*
by (*insert MWLsSteps-det.simps*[*of* $skip_\iota$ *m1*],
force)

have *nextm1'*:
(*THE m2'*. ($\exists$ *p* α. $\langle skip_\iota$,*m1'*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p*,*m2'*$\rangle$)) = *m1'*
by (*insert MWLsSteps-det.simps*[*of* $skip_\iota$ *m1'*],
force)

with *dequal nextm1* **show**
THE m2. $\exists$ *p* α. $\langle skip_\iota$,*m1*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p*,*m2*$\rangle$ $=_d$
THE m2'. $\exists$ *p* α. $\langle skip_\iota$,*m1'*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p*,*m2'*$\rangle$
by *auto*
next
fix *p m1 m1' m2* α
assume *dequal*: *m1* $\sim_{d,(htchLocSet\ PP)}$ *m1'*
assume *skipstep*: $\langle skip_\iota$,*m1*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p*,*m2*$\rangle$
with *MWLsSteps-det.simps*[*of* $skip_\iota$ *m1* α *p m2*]
have *aux*: *p* = *None* $\wedge$ *m2* = *m1* $\wedge$ α = []
by *auto*
with *dequal skipstep MWLsSteps-det.skip*
show $\exists$ *p' m2'*.
$\langle skip_\iota$,*m1'*$\rangle$ $\rightarrow\lhd\alpha\rhd$ $\langle$*p'*,*m2'*$\rangle$ $\wedge$

stepResultsinR p p' $\{(V,\ V').\ V = V' \wedge$
$(V = [] \vee V = [skip_\iota])\} \wedge$
$(\alpha = [] \vee \alpha = [skip_\iota]) \wedge$
dhequality-alternative d PP ι *m2 m2'*
by (*simp add*: *stepResultsinR-def dhequality-alternative-def*,
fastforce)
qed

with *inR* **show** $\exists\, R.\ SdlHPPB\ d\ PP\ R \wedge ([skip_\iota],[skip_\iota]) \in R$
by *auto*
qed

— auxiliary lemma for assign side condition (lemma 9 in original paper)
lemma *semAssignSC-aux*:
assumes *dhind*: $e \equiv_{DA\ x,(htchLoc\ \iota)}\ e$
shows $NDC\ d\ (x :=_\iota e) \vee IDC\ d\ (x :=_\iota e)\ (htchLoc\ (pp\ (x{:=}_\iota\ e)))$
proof (*simp add*: *IDC-def*, *auto*, *simp add*: *NDC-def*)
fix *m1 m1'*
assume *dhequal*: $m1 \sim_{d,htchLoc\ \iota}\ m1'$
hence *dequal*: $m1 =_d m1'$
by (*simp add*: *dH-equal-def*)

obtain *v* **where** *veq*: $E\ e\ m1 = v$ **by** *auto*
hence *m2eq*: $[\![x :=_\iota e]\!](m1) = m1(x := v)$
by (*simp add*: *NextMem-def*,
insert MWLsSteps-det.simps[*of* $x :=_\iota e$ *m1*],
force)

obtain *v'* **where** *v'eq*: $E\ e\ m1' = v'$ **by** *auto*
hence *m2'eq*: $[\![x :=_\iota e]\!](m1') = m1'(x := v')$
by (*simp add*: *NextMem-def*,
insert MWLsSteps-det.simps[*of* $x :=_\iota e$ *m1'*],
force)

from *dhequal* **have** *shiftdomain*:
$DA\ x \leq d \Longrightarrow m1 \sim_{DA\ x,(htchLoc\ \iota)}\ m1'$
by (*simp add*: *dH-equal-def d-equal-def htchLoc-def*)

with *veq v'eq dhind*
have $(DA\ x) \leq d \Longrightarrow v = v'$
by (*simp add*: *dH-indistinguishable-def*)

with *dequal m2eq m2'eq*
show $[\![x :=_\iota e]\!](m1) =_d [\![x :=_\iota e]\!](m1')$
by (*simp add*: *d-equal-def*)
qed

theorem *WHATWHERE-Secure-Assign*:
assumes *dhind*: $e \equiv_{DA\ x,(htchLoc\ \iota)}\ e$
assumes *dheq-imp*: $\forall\, m\ m'\ d\ \iota'.\ (m \sim_{d,(htchLoc\ \iota')} m' \wedge$
$[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m'))$
$\longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m')$
shows *WHATWHERE-Secure* $[x :=_\iota e]$
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
fix *d PP*
def $R \equiv \{(V{::}('exp,'id)\ MWLsCom\ list, V'{::}('exp,'id)\ MWLsCom\ list).$
$V = V' \wedge (V = [] \vee V = [x :=_\iota e])\}$

have *inR*: $([x :=_\iota e],[x :=_\iota e]) \in R$
by (*simp add*: *R-def*)

have *SdlHPPB d PP R*
proof (*simp add*: *Strong-dlHPP-Bisimulation-def R-def*
sym-def trans-def, *auto*)
assume *notIDC*: $\neg$ *IDC d* $(x :=_\iota e)$ $(htchLoc\ \iota)$
with *dhind semAssignSC-aux*
show *NDC d* $(x :=_\iota e)$
by *force*
next
fix *m1 m1′ m2 p* α
assume *assignstep*: $\langle x :=_\iota e, m1\rangle \rightarrow\lhd\alpha\rhd\ \langle p, m2\rangle$
assume *dhequal*: $m1 \sim_{d,htchLocSet\ PP} m1'$

from *assignstep* **have** *nextm1*:
$p = None \wedge \alpha = [] \wedge [\![x :=_\iota e]\!](m1) = m2$
by (*simp add*: *NextMem-def*,
insert MWLsSteps-det.simps[*of* $x :=_\iota e$ *m1*], *force*)

obtain *m2′* **where** *nextm1′*:
$\langle x :=_\iota e, m1'\rangle \rightarrow\lhd[]\rhd\ \langle None, m2'\rangle \wedge [\![x :=_\iota e]\!](m1') = m2'$
by (*simp add*: *NextMem-def*,
insert MWLsSteps-det.simps[*of* $x :=_\iota e$ *m1′*], *force*)

from *dhequal* **have** *dhequal-ι*: $\bigwedge\iota.\ htchLoc\ \iota \subseteq htchLocSet\ PP$
$\Longrightarrow m1 \sim_{d,(htchLoc\ \iota)} m1'$
by (*simp add*: *dH-equal-def*, *auto*)

with *dhind semAssignSC-aux*
have $htchLoc\ \iota \subseteq htchLocSet\ PP \Longrightarrow$
$[\![x :=_\iota e]\!](m1) =_d [\![x :=_\iota e]\!](m1')$
by (*simp add*: *NDC-def IDC-def dH-equal-def*, *fastforce*)

with *dhind dheq-imp dhequal-ι*
have $htchLoc\ \iota \subseteq htchLocSet\ PP \Longrightarrow$
$[\![x :=_\iota e]\!](m1) \sim_{d,(htchLocSet\ PP)} [\![x :=_\iota e]\!](m1')$

by (*simp add*: *htchLocSet-def dH-equal-def*, *blast*)

with *nextm1 nextm1'* *assignstep dhequal*
show $\exists p'\ m2'.$
$\langle x :=_\iota e, m1' \rangle \rightarrow\triangleleft\alpha\triangleright \langle p', m2' \rangle \wedge$
$stepResultsinR\ p\ p'\ \{(V,\ V').\ V = V' \wedge (V = [] \vee V = [x{:=}_\iota\ e])\} \wedge$
$(\alpha = [] \vee \alpha = [x{:=}_\iota\ e]) \wedge dhequality\text{-}alternative\ d\ PP\ \iota\ m2\ m2'$
by (*auto simp add*: *stepResultsinR-def dhequality-alternative-def*)
qed

with *inR* **show** $\exists R.\ SdlHPPB\ d\ PP\ R \wedge ([x{:=}_\iota\ e],[x{:=}_\iota\ e]) \in R$
by *auto*
qed

end

end

3.3 Proofs of non-atomic compositionality results

We prove compositionality results for each non-atomic command of our example programming language (i.e. a command that is composed out of other commands): If the components are strongly secure and the expressions involved indistinguishable for an observer on security level d, then the composed command is also strongly secure.

theory *Language-Composition*
imports *WHATWHERE-Secure-Skip-Assign*
begin

context *WHATWHERE-Secure-Programs*
begin

theorem *Compositionality-Seq*:
assumes *WWs-part1*: *WHATWHERE-Secure* [*c1*]
assumes *WWs-part2*: *WHATWHERE-Secure* [*c2*]
assumes *uniPPc1c2*: *unique-PPc* (*c1*;*c2*)
shows *WHATWHERE-Secure* [*c1*;*c2*]
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
fix *d PP*

from *uniPPc1c2* **have** *nocommonPP*: $set\ (PPc\ c1) \cap set\ (PPc\ c2) = \{\}$
by (*simp add*: *unique-PPV-def unique-PPc-def*)

from *WWs-part1* **obtain** *R1'* **where** *R1'assump*:
$SdlHPPB\ d\ PP\ R1' \wedge ([c1],[c1]) \in R1'$
by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

def $R1 \equiv \{(V,V'). (V,V') \in R1' \wedge set\ (PPV\ V) \subseteq set\ (PPc\ c1)$
$\wedge\ set\ (PPV\ V') \subseteq set\ (PPc\ c1)\}$

from *R1'assump R1-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPR1*: *SdlHPPB d PP R1*
by *force*

from *WWs-part2* **obtain** *R2'* **where** *R2'assump*:
$SdlHPPB\ d\ PP\ R2' \wedge ([c2],[c2]) \in R2'$
by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

def $R2 \equiv \{(V,V'). (V,V') \in R2' \wedge set\ (PPV\ V) \subseteq set\ (PPc\ c2)$
$\wedge\ set\ (PPV\ V') \subseteq set\ (PPc\ c2)\}$

from *R2'assump R2-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPR2*: *SdlHPPB d PP R2*
by *force*

from *nocommonPP* **have** *nocommonDomain*: $Domain\ R1 \cap Domain\ R2 \subseteq \{[]\}$
by (*simp add*: *R1-def R2-def*, *auto*,
metis inf-greatest inf-idem le-bot unique-V-uneq)

with *commonArefl-subset-commonDomain*
have *Areflassump1*: $Arefl\ R1 \cap Arefl\ R2 \subseteq \{[]\}$
by *force*

def $R0 \equiv \{(s1,s2). \exists\, c1\ c1'\ c2\ c2'.\ s1 = [c1;c2] \wedge s2 = [c1';c2'] \wedge$
$([c1],[c1']) \in R1 \wedge ([c2],[c2']) \in R2\}$

with *R1-def R1'assump R2-def R2'assump*
have *inR0*: $([c1;c2],[c1;c2]) \in R0$
by *auto*

have $Domain\ R0 \cap Domain\ (R1 \cup R2) = \{\}$
by (*simp add*: *R0-def R1-def R2-def*, *auto*, *metis Int-absorb1 Int-assoc Int-empty-left*

nocommonPP unique-c-uneq, *metis Int-absorb Int-absorb1*
Int-assoc Int-empty-left nocommonPP unique-c-uneq)

with *commonArefl-subset-commonDomain*
have *Areflassump2*: $Arefl\ R0 \cap Arefl\ (R1 \cup R2) \subseteq \{[]\}$
by *force*

have *disjuptoR0*:
$disj\text{-}dlHPP\text{-}Bisimulation\text{-}Up\text{-}To\text{-}R'\ d\ PP\ (R1 \cup R2)\ R0$
proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-R'-def*, *auto*)
from *Areflassump1 SdlHPPR1 SdlHPPR2 Union-Strong-dlHPP-Bisim*
show $SdlHPPB\ d\ PP\ (R1 \cup R2)$

```
    by metis
  next
    from SdlHPPR1 have symR1: sym R1
      by (simp add: Strong-dlHPP-Bisimulation-def)
    from SdlHPPR2 have symR2: sym R2
      by (simp add: Strong-dlHPP-Bisimulation-def)
    with symR1 R0-def show sym R0
      by (simp add: sym-def, auto)
  next
    from SdlHPPR1 have transR1: trans R1
      by (simp add: Strong-dlHPP-Bisimulation-def)
    from SdlHPPR2 have transR2: trans R2
      by (simp add: Strong-dlHPP-Bisimulation-def)
    show trans R0
      proof -
        {
        fix V V' V''
        assume p1: (V,V') ∈ R0
        assume p2: (V',V'') ∈ R0
        have (V,V'') ∈ R0
          proof -
            from p1 R0-def obtain c1 c2 c1' c2' where p1assump:
              V = [c1;c2] ∧ V' = [c1';c2'] ∧
              ([c1],[c1']) ∈ R1 ∧ ([c2],[c2']) ∈ R2
              by auto
            with p2 R0-def obtain c1'' c2'' where p2assump:
              V'' = [c1'';c2''] ∧
              ([c1'],[c1'']) ∈ R1 ∧ ([c2'],[c2'']) ∈ R2
              by auto
            with p1assump transR1 transR2 have
              trans-assump: ([c1],[c1'']) ∈ R1 ∧ ([c2],[c2'']) ∈ R2
              by (simp add: trans-def, blast)
            with p1assump p2assump R0-def show ?thesis
              by auto
          qed
        }
        thus ?thesis unfolding trans-def by blast
      qed
  next
    fix V V'
    assume (V,V') ∈ R0
    with R0-def show length V = length V'
      by auto
  next
    fix V V' i
    assume inR0: (V,V') ∈ R0
    assume irange: i < length V
    assume notIDC: ¬ IDC d (V!i) (htchLoc (pp (V!i)))
    from inR0 R0-def obtain c1 c2 c1' c2' where VV'assump:
```

$V = [c1;c2] \wedge V' = [c1';c2'] \wedge$
$([c1],[c1']) \in R1 \wedge ([c2],[c2']) \in R2$
by *auto*
have *eqnextmem*: $\bigwedge m.\ [\![c1;c2]\!](m) = [\![c1]\!](m)$
proof −
fix *m*
from *nextmem-exists-and-unique* **obtain** m' **where** *c1nextmem*:
$\exists\, p\ \alpha.\ \langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m''\rangle) \longrightarrow m'' = m')$
by *force*

hence *eqdir1*: $[\![c1]\!](m) = m'$
by (*simp add*: *NextMem-def*, *auto*)

from *c1nextmem* **obtain** $p\ \alpha$ **where** $\langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
by *auto*

with *c1nextmem* **have** $\exists\, p\ \alpha.\ \langle c1;c2,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle c1;c2,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m''\rangle) \longrightarrow m'' = m')$
by (*auto*, *metis MWLsSteps-det.seq1 MWLsSteps-det.seq2*
option.exhaust, *metis MWLsSteps-det-cases*(*3*))

hence *eqdir2*: $[\![c1;c2]\!](m) = m'$
by (*simp add*: *NextMem-def*, *auto*)

with *eqdir1* **show** $[\![c1;c2]\!](m) = [\![c1]\!](m)$
by *auto*
qed

have *eqpp*: *pp* (*c1*;*c2*) = *pp c1*
by *simp*
from *VV'assump SdlHPPR1* **have** *IDC d c1* (*htchLoc* (*pp c1*))
$\vee$ *NDC d c1*
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
with *eqnextmem eqpp* **have** *IDC d* (*c1*;*c2*)
(*htchLoc* (*pp* (*c1*;*c2*))) $\vee$ *NDC d* (*c1*;*c2*)
by (*simp add*: *IDC-def NDC-def*)
with *inR0 irange notIDC VV'assump*
show *NDC d* (*V*!*i*)
by (*simp add*: *IDC-def*, *auto*)
next
fix $V\ V'\ m1\ m1'\ m2\ \alpha\ p\ i$
assume *inR0*: $(V,V') \in R0$
assume *irange*: $i < length\ V$
assume *step*: $\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,htchLocSet\ PP} m1'$

from *inR0 R0-def* **obtain** $c1\ c1'\ c2\ c2'$ **where** *R0pair*:
$V = [c1;c2] \wedge V' = [c1';c2'] \wedge ([c1],[c1']) \in R1$

```
  ∧ ([c2],[c2']) ∈ R2
  by auto

from R0pair irange have i0: i = 0 by simp

have eqpp: pp (c1;c2) = pp c1
 by simp

— get the two different cases:
from R0pair step i0 obtain c3 where case-distinction:
  (p = Some c2 ∧ ⟨c1,m1⟩ →⊲α⊳ ⟨None,m2⟩)
  ∨ (p = Some (c3;c2) ∧ ⟨c1,m1⟩ →⊲α⊳ ⟨Some c3,m2⟩)
  by (simp, insert MWLsSteps-det.simps[of c1;c2 m1],
    auto)
moreover
— Case 1: first command terminates
{
  assume passump: p = Some c2
  assume StepAssump: ⟨c1,m1⟩ →⊲α⊳ ⟨None,m2⟩
  hence Vstep-case1:
    ⟨c1;c2,m1⟩ →⊲α⊳ ⟨Some c2,m2⟩
    by (simp add: MWLsSteps-det.seq1)

  from SdlHPPR1 StepAssump R0pair dhequal
    strongdlHPPB-aux[of d PP
    R1 0 [c1] [c1'] m1 α None m2 m1']
  obtain p' α' m2' where c1c1'reason:
    p' = None ∧ ⟨c1',m1'⟩ →⊲α'⊳ ⟨p',m2'⟩ ∧ (α,α') ∈ R1 ∧
    dhequality-alternative d PP (pp c1) m2 m2'
    by (simp add: stepResultsinR-def, fastforce)

  with eqpp c1c1'reason have conclpart:
    ⟨c1';c2',m1'⟩ →⊲α'⊳ ⟨Some c2',m2'⟩ ∧
    dhequality-alternative d PP (pp (c1;c2)) m2 m2'
    by (auto, simp add: MWLsSteps-det.seq1)

  with passump R0pair c1c1'reason i0
  have case1-concl:
  ∃ p' α' m2'.
     ⟨V'!i,m1'⟩ →⊲α'⊳ ⟨p',m2'⟩ ∧
     stepResultsinR p p' (R0 ∪ (R1 ∪ R2)) ∧
     ((α,α') ∈ R0 ∨ (α,α') ∈ R1 ∨ (α,α') ∈ R2) ∧
     dhequality-alternative d PP (pp (V!i)) m2 m2'
  by (simp add: stepResultsinR-def, auto)
}
moreover
— Case 2: first command does not terminate
{
  assume passump: p = Some (c3;c2)
```

assume *StepAssump*: $\langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ c3,m2\rangle$

hence *Vstep-case2*: $\langle c1;c2,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ (c3;c2),m2\rangle$
by (*simp add*: *MWLsSteps-det.seq2*)

from *SdlHPPR1 StepAssump R0pair dhequal*
strongdlHPPB-aux[*of d PP*
R1 0 [*c1*] [*c1'*] *m1* α *Some c3 m2 m1'*]
obtain p' $c3'$ α' $m2'$ **where** *c1c1'reason*:
$p' = Some\ c3' \wedge \langle c1',m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$([c3],[c3']) \in R1 \wedge (\alpha,\alpha') \in R1 \wedge$
dhequality-alternative d PP (*pp c1*) *m2 m2'*
by (*simp add*: *stepResultsinR-def*, *fastforce*)

with *eqpp c1c1'reason* **have** *conclpart*:
$\langle c1';c2',m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle Some\ (c3';c2'),m2'\rangle \wedge$
dhequality-alternative d PP (*pp* (*c1*;*c2*)) *m2 m2'*
by (*auto*, *simp add*: *MWLsSteps-det.seq2*)

from *c1c1'reason R0pair R0-def* **have**
$([c3;c2],[c3';c2']) \in R0$
by *auto*

with *R0pair conclpart passump c1c1'reason i0*
have *case1-concl*:
$\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
stepResultsinR p p' $(R0 \cup (R1 \cup R2)) \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R1 \vee (\alpha,\alpha') \in R2) \wedge$
dhequality-alternative d PP (*pp* ($V!i$)) *m2 m2'*
by (*simp add*: *stepResultsinR-def*, *auto*)
}
ultimately
show $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
stepResultsinR p p' $(R0 \cup (R1 \cup R2)) \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R1 \vee (\alpha,\alpha') \in R2) \wedge$
dhequality-alternative d PP (*pp* ($V!i$)) *m2 m2'*
by *blast*
qed

with *inR0 Areflassump2 Up-To-Technique*
have *SdlHPPB d PP* $(R0 \cup (R1 \cup R2))$
by *auto*

with *inR0* **show** $\exists R.\ SdlHPPB\ d\ PP\ R \wedge ([c1;c2],[c1;c2]) \in R$
by *auto*

qed

theorem *Compositionality-Spawn*:
  assumes *WWs-threads*: *WHATWHERE-Secure V*
  assumes *uniPPspawn*: *unique-PPc* ($spawn_\iota$ *V*)
  shows *WHATWHERE-Secure* [$spawn_\iota$ *V*]
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
  fix *d PP*

  from *uniPPspawn* **have** *pp-difference*: $\iota \notin$ *set* (*PPV V*)
    by (*simp add*: *unique-PPc-def*)

  — Step 1
  from *WWs-threads* **obtain** R' **where** R'*assump*:
    SdlHPPB d PP $R' \wedge (V,V) \in R'$
    by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

  def $R \equiv \{(V',V'').\ (V',V'') \in R' \wedge set\ (PPV\ V') \subseteq set\ (PPV\ V)$
    $\wedge\ set\ (PPV\ V'') \subseteq set\ (PPV\ V)\}$

  from R'*assump R-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
  have *SdlHPPR*: *SdlHPPB d PP R*
    by *force*

  — Step 2
  def $R0 \equiv \{(sp1,sp2).\ \exists \iota'\ \iota''\ V'\ V''.$
    $sp1 = [spawn_{\iota'}\ V'] \wedge sp2 = [spawn_{\iota''}\ V'']$
    $\wedge\ \iota' \notin set\ (PPV\ V) \wedge \iota'' \notin set\ (PPV\ V) \wedge (V',V'') \in R\}$

  with *R-def* R'*assump pp-difference* **have** *inR0*:
    $([spawn_\iota\ V],[spawn_\iota\ V]) \in R0$
    by *auto*

  — Step 3
  from *R0-def R-def* R'*assump* **have** *Domain R0* $\cap$ *Domain R* $= \{\}$
    by *auto*

  with *commonArefl-subset-commonDomain*
  have *Areflassump*: *Arefl R0* $\cap$ *Arefl R* $\subseteq \{[]\}$
    by *force*

  — Step 4
  have *disjuptoR0*:
    disj-dlHPP-Bisimulation-Up-To-R' *d PP R R0*
    proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-*R'*-def*, *auto*)
      from *SdlHPPR* **show** *SdlHPPB d PP R*
        by *auto*
    next
      from *SdlHPPR* **have** *symR*: *sym R*
        by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
      with *R0-def* **show** *sym R0*

by (*simp add*: *sym-def*, *auto*)
next
from *SdlHPPR* **have** *transR*: *trans R*
by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
with *R0-def* **show** *trans R0*
proof −
{
fix *V1 V2 V3*
assume *inR1*: $(V1,V2) \in R0$
assume *inR2*: $(V2,V3) \in R0$
from *inR1 R0-def* **obtain** $W\ W'\ \iota\ \iota'$ **where** *p1*: $V1 = [spawn_{\iota}\ W]$
$\wedge\ V2 = [spawn_{\iota'}\ W'] \wedge \iota \notin set\ (PPV\ V) \wedge \iota' \notin set\ (PPV\ V)$
$\wedge\ (W,W') \in R$
by *auto*
with *inR2 R0-def* **obtain** $W''\ \iota''$ **where** *p2*: $V3 = [spawn_{\iota''}\ W'']$
$\wedge\ \iota'' \notin set\ (PPV\ V) \wedge (W',W'') \in R$
by *auto*
from *p1 p2 transR* **have** $(W,W'') \in R$
by (*simp add*: *trans-def*, *auto*)
with *p1 p2 R0-def* **have** $(V1,V3) \in R0$
by *auto*
}
thus *?thesis* **unfolding** *trans-def* **by** *blast*
qed
next
fix $V'\ V''$
from *SdlHPPR* **have** *eqlenR*: $(V',V'') \in R \Longrightarrow length\ V' = length\ V''$
by (*simp add*: *Strong-dlHPP-Bisimulation-def*, *auto*)
with *R0-def* **show** $(V',V'') \in R0 \Longrightarrow length\ V' = length\ V''$
by *auto*
next
fix $V'\ V''\ i$
assume *inR0*: $(V',V'') \in R0$
assume *irange*: $i < length\ V'$
from *inR0 R0-def* **obtain** $\iota'\ \iota''\ W'\ W''$
where *R0pair*: $V' = [spawn_{\iota'}\ W'] \wedge V'' = [spawn_{\iota''}\ W'']$
by *auto*
{
fix *m*
from *nextmem-exists-and-unique* **obtain** m' **where** *spawnnextmem*:
$\exists p\ \alpha.\ \langle spawn_{\iota'}\ W',m\rangle \rightarrow\lhd\alpha\rhd \langle p,m'\rangle$
$\wedge\ (\forall m''.\ (\exists p\ \alpha.\ \langle spawn_{\iota'}\ W',m\rangle \rightarrow\lhd\alpha\rhd \langle p,m''\rangle) \longrightarrow m'' = m')$
by *force*

hence $m = m'$
by (*metis MWLsSteps-det.spawn*)

with *spawnnextmem* **have** *eqnextmem*:
$[\![spawn_{\iota'}\ W']\!](m) = m$

by (*simp add*: *NextMem-def*, *auto*)
}

with *R0pair irange* **show** *NDC d* ($V'!i$)
by (*simp add*: *NDC-def*)
next
fix V' V'' *i m1 m1'* *m2* α *p*
assume *inR0*: $(V',V'') \in R0$
assume *irange*: $i <$ *length* V'
assume *step*: $\langle V'!i,m1\rangle \rightarrow\lhd\alpha\rhd \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,htchLocSet\ PP} m1'$

from *inR0 R0-def* **obtain** ι' ι'' W' W''
where *R0pair*: $V' = [spawn_{\iota'}\ W']$
$\wedge\ V'' = [spawn_{\iota''}\ W''] \wedge (W',W'') \in R$
by *auto*

with *step irange*
have *conc-step1*: $\alpha = W' \wedge p = None \wedge m2 = m1$
by (*simp*, *metis MWLsSteps-det-cases*(*6*))

from *R0pair irange*
obtain p' α' $m2'$ **where** *conc-step2*: $\langle V''!i,m1'\rangle \rightarrow\lhd\alpha'\rhd \langle p',m2'\rangle$
$\wedge\ p' = None \wedge \alpha' = W'' \wedge m2' = m1'$
by (*simp*, *metis MWLsSteps-det.spawn*)

with *R0pair conc-step1 dhequal irange*
show $\exists\, p'\ \alpha'\ m2'.\ \langle V''!i,m1'\rangle \rightarrow\lhd\alpha'\rhd \langle p',m2'\rangle\ \wedge$
stepResultsinR p p' $(R0 \cup R)\ \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R)\ \wedge$
dhequality-alternative d PP (*pp* ($V'!i$)) *m2* $m2'$
by (*simp add*: *stepResultsinR-def*
dhequality-alternative-def, *auto*)
qed
— Step 5
with *Areflassump Up-To-Technique*
have *SdlHPPB d PP* $(R0 \cup R)$
by *auto*

with *inR0* **show** $\exists\, R.$ *SdlHPPB d PP R* $\wedge$
$([spawn_\iota\ V],[spawn_\iota\ V]) \in R$
by *auto*
qed

theorem *Compositionality-If*:
assumes *dind*: $\forall\, d.\ b \equiv_d b$
assumes *WWs-branch1*: *WHATWHERE-Secure* [*c1*]
assumes *WWs-branch2*: *WHATWHERE-Secure* [*c2*]

assumes *uniPPif*: *unique-PPc* (if_{ι} *b then c1 else c2 fi*)
shows *WHATWHERE-Secure* [if_{ι} *b then c1 else c2 fi*]
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
fix *d PP*
from *uniPPif* **have** *nocommonPP*: *set* (*PPc c1*) ∩ *set* (*PPc c2*) = {}
by (*simp add*: *unique-PPc-def*)

from *uniPPif* **have** *pp-difference*: $\iota \notin$ *set* (*PPc c1*) ∪ *set* (*PPc c2*)
by (*simp add*: *unique-PPc-def*)

from *WWs-branch1* **obtain** *R1'* **where** *R1'assump*:
SdlHPPB d PP R1' ∧ ([*c1*],[*c1*]) ∈ *R1'*
by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

def *R1* ≡ {(*V*,*V'*). (*V*,*V'*) ∈ *R1'* ∧ *set* (*PPV V*) ⊆ *set* (*PPc c1*)
∧ *set* (*PPV V'*) ⊆ *set* (*PPc c1*)}

from *R1'assump R1-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPR1*: *SdlHPPB d PP R1*
by *force*

from *WWs-branch2* **obtain** *R2'* **where** *R2'assump*:
SdlHPPB d PP R2' ∧ ([*c2*],[*c2*]) ∈ *R2'*
by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

def *R2* ≡ {(*V*,*V'*). (*V*,*V'*) ∈ *R2'* ∧ *set* (*PPV V*) ⊆ *set* (*PPc c2*)
∧ *set* (*PPV V'*) ⊆ *set* (*PPc c2*)}

from *R2'assump R2-def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPR2*: *SdlHPPB d PP R2*
by *force*

from *nocommonPP* **have** *Domain R1* ∩ *Domain R2* ⊆ {[]}

by (*simp add*: *R1-def R2-def*, *auto*,
metis empty-subsetI inf-idem inf-mono set-eq-subset unique-V-uneq)

with *commonArefl-subset-commonDomain*
have *Areflassump1*: *Arefl R1* ∩ *Arefl R2* ⊆ {[]}
by *force*

with *SdlHPPR1 SdlHPPR2 Union-Strong-dlHPP-Bisim* **have** *SdlHPPR1R2*:
SdlHPPB d PP (*R1* ∪ *R2*)
by *force*

def *R* ≡ (*R1* ∪ *R2*) ∪ {([],[])}

def *R0* ≡ {(*i1*,*i2*). $\exists \iota' \, \iota'' \, b' \, b''$ *c1' c1'' c2' c2''*.
i1 = [$if_{\iota'}$ *b' then c1' else c2' fi*]

$\wedge$ *i2* = $[if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi]$
$\wedge\ \iota' \notin (set\ (PPc\ c1) \cup set\ (PPc\ c2))$
$\wedge\ \iota'' \notin (set\ (PPc\ c1) \cup set\ (PPc\ c2)) \wedge ([c1'],[c1'']) \in R1$
$\wedge\ ([c2'],[c2'']) \in R2 \wedge b' \equiv_d b''\}$

with *R-def R1-def R1'assump R2-def R2'assump pp-difference dind*
have *inR0*: $([if_{\iota}\ b\ then\ c1\ else\ c2\ fi],[if_{\iota}\ b\ then\ c1\ else\ c2\ fi]) \in R0$
  by *auto*

from *R0-def R-def R1-def R2-def*
have *Domain R0* $\cap$ *Domain R* = {}
  by *auto*

with *commonArefl-subset-commonDomain*
have *Areflassump2*: *Arefl R0* $\cap$ *Arefl R* $\subseteq$ {[]}
  by *force*

have *disjuptoR0*:
  disj-dlHPP-Bisimulation-Up-To-R' d PP R R0
  proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-R'-def*, *auto*)
    from *SdlHPPR1R2 adding-emptypair-keeps-SdlHPPB*
    show *SdlHPPB d PP R*
      by (*simp add*: *R-def*)
  next
    from *SdlHPPR1* **have** *symR1*: *sym R1*
      by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
    from *SdlHPPR2* **have** *symR2*: *sym R2*
      by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
    from *symR1 symR2 d-indistinguishable-sym R0-def* **show** *sym R0*
      by (*simp add*: *sym-def*, *fastforce*)
  next
    from *SdlHPPR1* **have** *transR1*: *trans R1*
      by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
    from *SdlHPPR2* **have** *transR2*: *trans R2*
      by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
    show *trans R0*
      proof −
        {
          fix $V'\ V''\ V'''$
          assume *p1*: $(V',V'') \in R0$
          assume *p2*: $(V'',V''') \in R0$

          from *p1 R0-def* **obtain** $\iota'\ \iota''\ b'\ b''\ c1'\ c1''\ c2'\ c2''$ **where**
            passump1: $V' = [if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi]$
            $\wedge\ V'' = [if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi]$
            $\wedge\ \iota' \notin (set\ (PPc\ c1) \cup set\ (PPc\ c2))$
            $\wedge\ \iota'' \notin (set\ (PPc\ c1) \cup set\ (PPc\ c2))$
            $\wedge\ ([c1'],[c1'']) \in R1 \wedge ([c2'],[c2'']) \in R2$
            $\wedge\ b' \equiv_d b''$

by *force*

with *p2 R0-def* **obtain** ι''' *b''' c1''' c2'''* **where**
passump2: $V''' = [if_{\iota'''}\ b'''\ then\ c1'''\ else\ c2'''\ fi]$
$\wedge\ \iota''' \notin (set\ (PPc\ c1) \cup set\ (PPc\ c2))$
$\wedge\ ([c1''],[c1''']) \in R1 \wedge ([c2''],[c2''']) \in R2$
$\wedge\ b'' \equiv_d b'''$
by *force*

with *passump1 transR1 transR2 d-indistinguishable-trans*
have $([c1'],[c1''']) \in R1 \wedge ([c2'],[c2''']) \in R2$
$\wedge\ b' \equiv_d b'''$
by (*metis transD*)

with *passump1 passump2 R0-def* **have** $(V',V''') \in R0$
by *auto*
}
thus *?thesis* **unfolding** *trans-def* **by** *blast*
qed
next
fix $V\ V'$
assume *inR0*: $(V,V') \in R0$
with *R0-def* **show** *length* V = *length* V' **by** *auto*
next
fix $V'\ V''\ i$
assume *inR0*: $(V',V'') \in R0$
assume *irange*: $i <$ *length* V'
assume *notIDC*: $\neg$ *IDC d* $(V'!i)$ (*htchLoc* (*pp* $(V'!i)$))

from *inR0 R0-def* **obtain** $\iota'\ \iota''\ b'\ b''\ c1'\ c1''\ c2'\ c2''$
where *R0pair*: $V' = [if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi]$
$\wedge\ V'' = [if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi]$
$\wedge\ \iota' \notin set\ (PPc\ c1) \cup set\ (PPc\ c2)$
$\wedge\ \iota'' \notin set\ (PPc\ c1) \cup set\ (PPc\ c2)$
$\wedge\ ([c1'],[c1'']) \in R1 \wedge ([c2'],[c2'']) \in R2$
$\wedge\ b' \equiv_d b''$
by *force*

have *NDC d* $(if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi)$
proof $-$
{
fix m
from *nextmem-exists-and-unique* **obtain** $m'\ p\ \alpha$ **where** *ifnextmem*:
$\langle if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi,m\rangle \rightarrow\lhd\alpha\rhd \langle p,m'\rangle$
$\wedge\ (\forall m''.(\exists p\ \alpha.\langle if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi,m\rangle \rightarrow\lhd\alpha\rhd \langle p,m''\rangle)$
$\longrightarrow m'' = m')$
by *blast*

hence $m = m'$

by (*metis MWLsSteps-det.iffalse MWLsSteps-det.iftrue*)

with *ifnextmem* **have** *eqnextmem*:
$[\![if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi]\!](m) = m$
by (*simp add*: *NextMem-def*, *auto*)
}
thus *?thesis*
by (*simp add*: *NDC-def*)
qed

with *R0pair irange* **show** $NDC\ d\ (V'!i)$
by *simp*
next
fix $V'\ V''\ i\ m1\ m1'\ m2\ \alpha\ p$
assume *inR0*: $(V',V'') \in R0$
assume *irange*: $i < length\ V'$
assume *step*: $\langle V'!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright\ \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,htchLocSet\ PP}\ m1'$

from *inR0 R0-def* **obtain** $\iota'\ \iota''\ b'\ b''\ c1'\ c1''\ c2'\ c2''$
where *R0pair*: $V' = [if_{\iota'}\ b'\ then\ c1'\ else\ c2'\ fi]$
$\wedge\ V'' = [if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi]$
$\wedge\ \iota' \notin set\ (PPc\ c1) \cup set\ (PPc\ c2)$
$\wedge\ \iota'' \notin set\ (PPc\ c1) \cup set\ (PPc\ c2)$
$\wedge\ ([c1'],[c1'']) \in R1\ \wedge\ ([c2'],[c2'']) \in R2$
$\wedge\ b' \equiv_d b''$
by *force*

with *R0pair irange step* **have** *case-distinction*:
$(p = Some\ c1' \wedge BMap\ (E\ b'\ m1) = True)$
$\vee\ (p = Some\ c2' \wedge BMap\ (E\ b'\ m1) = False)$
by (*simp*, *metis MWLsSteps-det-cases*(*4*))
moreover
— Case 1: b evaluates to True
{
assume *passump*: $p = Some\ c1'$
assume *eval*: $BMap\ (E\ b'\ m1) = True$

from *R0pair step irange* **have** *stepconcl*: $\alpha = []\ \wedge\ m2 = m1$
by (*simp*, *metis MWLs-semantics.MWLsSteps-det-cases*(*4*))

from *eval R0pair dhequal* **have** *eval'*: $BMap\ (E\ b''\ m1') = True$
by (*simp add*: *d-indistinguishable-def dH-equal-def*, *auto*)

hence *step'*: $\langle if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi,m1'\rangle \rightarrow\triangleleft[]\triangleright$
$\langle Some\ c1'',m1'\rangle$
by (*simp add*: *MWLsSteps-det.iftrue*)

with *passump R0pair R-def dhequal stepconcl irange*

have $\exists\, p'\ \alpha'\ m2'.\ \langle V''!i, m1'\rangle \rightarrow\triangleleft\alpha'\triangleright\ \langle p', m2'\rangle\ \wedge$
stepResultsinR p p' $(R0 \cup R)\ \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R)\ \wedge$
dhequality-alternative d PP $(pp\ (V'!i))\ m2\ m2'$
by (*simp add*: *stepResultsinR-def dhequality-alternative-def*,
auto)
}
moreover
— Case 2: b evaluates to False
{
assume *passump*: $p = Some\ c2'$
assume *eval*: $BMap\ (E\ b'\ m1) = False$

from *R0pair step irange* **have** *stepconcl*: $\alpha = []\ \wedge\ m2 = m1$
by (*simp*, *metis MWLs-semantics.MWLsSteps-det-cases(4)*)

from *eval R0pair dhequal* **have** *eval'*: $BMap\ (E\ b''\ m1') = False$
by (*simp add*: *d-indistinguishable-def dH-equal-def*, *auto*)

hence *step'*: $\langle if_{\iota''}\ b''\ then\ c1''\ else\ c2''\ fi, m1'\rangle \rightarrow\triangleleft[]\triangleright$
$\langle Some\ c2'', m1'\rangle$
by (*simp add*: *MWLsSteps-det.iffalse*)

with *passump R0pair R-def dhequal stepconcl irange*
have $\exists\, p'\ \alpha'\ m2'.\ \langle V''!i, m1'\rangle \rightarrow\triangleleft\alpha'\triangleright\ \langle p', m2'\rangle\ \wedge$
stepResultsinR p p' $(R0 \cup R)\ \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R)\ \wedge$
dhequality-alternative d PP $(pp\ (V'!i))\ m2\ m2'$
by (*simp add*: *stepResultsinR-def dhequality-alternative-def*,
auto)
}
ultimately
show $\exists\, p'\ \alpha'\ m2'.\ \langle V''!i, m1'\rangle \rightarrow\triangleleft\alpha'\triangleright\ \langle p', m2'\rangle\ \wedge$
stepResultsinR p p' $(R0 \cup R)\ \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R)\ \wedge$
dhequality-alternative d PP $(pp\ (V'!i))\ m2\ m2'$
by *auto*
qed

with *Areflassump2 Up-To-Technique*
have *SdlHPPB d PP* $(R0 \cup R)$
by *auto*

with *inR0* **show** $\exists\, R.\ SdlHPPB\ d\ PP\ R$
$\wedge\ ([if_{\iota}\ b\ then\ c1\ else\ c2\ fi],[if_{\iota}\ b\ then\ c1\ else\ c2\ fi]) \in R$
by *auto*

qed

theorem *Compositionality-While*:
  assumes *dind*: $\forall\, d.\ b \equiv_d b$
  assumes *WWs-body*: *WHATWHERE-Secure* $[c]$
  assumes *uniPPwhile*: *unique-PPc* $(while_\iota\ b\ do\ c\ od)$
  shows *WHATWHERE-Secure* $[while_\iota\ b\ do\ c\ od]$
proof (*simp add*: *WHATWHERE-Secure-def*, *auto*)
  fix *d PP*

  from *uniPPwhile* **have** *pp-difference*: $\iota \notin set\ (PPc\ c)$
    by (*simp add*: *unique-PPc-def*)

  from *WWs-body* **obtain** R' **where** $R'assump$:
    $SdlHPPB\ d\ PP\ R' \wedge ([c],[c]) \in R'$
    by (*simp add*: *WHATWHERE-Secure-def*, *auto*)

  — add the empty pair because it is needed later in the proof
  def $R \equiv \{(V,V').\ (V,V') \in R' \wedge set\ (PPV\ V) \subseteq set\ (PPc\ c) \wedge$
    $set\ (PPV\ V') \subseteq set\ (PPc\ c)\} \cup \{([],[])\}$

  with $R'assump$ *SdlHPPB-restricted-on-PP-is-SdlHPPB*
    adding-emptypair-keeps-SdlHPPB
  have *SdlHPPR*: *SdlHPPB d PP R*
    proof −
      from $R'assump$ *SdlHPPB-restricted-on-PP-is-SdlHPPB* **have**
        SdlHPPB d PP
          $\{(V,V').\ (V,V') \in R' \wedge set\ (PPV\ V) \subseteq set\ (PPc\ c) \wedge$
          $set\ (PPV\ V') \subseteq set\ (PPc\ c)\}$
        by *force*

      with *adding-emptypair-keeps-SdlHPPB* **have**
        SdlHPPB d PP
          $(\{(V,V').\ (V,V') \in R' \wedge set\ (PPV\ V) \subseteq set\ (PPc\ c) \wedge$
          $set\ (PPV\ V') \subseteq set\ (PPc\ c)\} \cup \{([],[])\})$
        by *auto*

      with *R-def* **show** *?thesis*
        by *auto*
    qed

  def $R1 \equiv \{(w1,w2).\ \exists\,\iota\ \iota'\ b\ b'\ c1\ c1'\ c2\ c2'.$
    $w1 = [c1;(while_\iota\ b\ do\ c2\ od)]$
    $\wedge\ w2 = [c1';(while_{\iota'}\ b'\ do\ c2'\ od)]$
    $\wedge\ \iota \notin set\ (PPc\ c) \wedge \iota' \notin set\ (PPc\ c)$
    $\wedge\ ([c1],[c1']) \in R \wedge ([c2],[c2']) \in R$
    $\wedge\ b \equiv_d b'\}$

  def $R2 \equiv \{(w1,w2).\ \exists\,\iota\ \iota'\ b\ b'\ c1\ c1'.$
    $w1 = [while_\iota\ b\ do\ c1\ od]$
    $\wedge\ w2 = [while_{\iota'}\ b'\ do\ c1'\ od]$

$\wedge\ \iota \notin set\ (PPc\ c) \wedge \iota' \notin set\ (PPc\ c) \wedge$
$([c1],[c1']) \in R \wedge b \equiv_d b'\}$

def $R0 \equiv R1 \cup R2$

from *R2-def R-def R'assump pp-difference dind*
have *inR2*: $([while_\iota\ b\ do\ c\ od],[while_\iota\ b\ do\ c\ od]) \in R2$
  by *auto*

from *R0-def R1-def R2-def R-def R'assump* **have**
  $Domain\ R0 \cap Domain\ R = \{\}$
  by *auto*

with *commonArefl-subset-commonDomain*
have *Areflassump*: $Arefl\ R0 \cap Arefl\ R \subseteq \{[]\}$
  by *force*

— show some facts about R1 and R2 needed later in the proof at several positions

from *SdlHPPR* **have** *symR*: *sym R*
  by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
from *symR R1-def d-indistinguishable-sym* **have** *symR1*: *sym R1*
  by (*simp add*: *sym-def*, *fastforce*)
from *symR R2-def d-indistinguishable-sym* **have** *symR2*: *sym R2*
  by (*simp add*: *sym-def*, *fastforce*)

have *disjuptoR1R2*:
  disj-dlHPP-Bisimulation-Up-To-R' d PP R R0
proof (*simp add*: *disj-dlHPP-Bisimulation-Up-To-R'-def*, *auto*)
    from *SdlHPPR* **show** *SdlHPPB d PP R*
      by *auto*
  next
    from *symR1 symR2 R0-def* **show** *sym R0*
      by (*simp add*: *sym-def*)
  next
    from *SdlHPPR* **have** *transR*: *trans R*
      by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
    have *transR1*: *trans R1*
      proof −
        {
          fix $V\ V'\ V''$
          assume *p1*: $(V,V') \in R1$
          assume *p2*: $(V',V'') \in R1$

          from *p1 R1-def* **obtain** $\iota\ \iota'\ b\ b'\ c1\ c1'\ c2\ c2'$ **where** *passump1*:
            $V = [c1;(while_\iota\ b\ do\ c2\ od)]$
            $\wedge\ V' = [c1';(while_{\iota'}\ b'\ do\ c2'\ od)]$
            $\wedge\ \iota \notin set\ (PPc\ c) \wedge \iota' \notin set\ (PPc\ c)$
            $\wedge\ ([c1],[c1']) \in R \wedge ([c2],[c2']) \in R$

$\wedge\ b \equiv_d b'$
by *force*

with *p2 R1-def* **obtain** $\iota''\ b''\ c1''\ c2''$ **where** *passump2*:
$V'' = [c1'';(while_{\iota''}\ b''\ do\ c2''\ od)] \wedge \iota'' \notin set\ (PPc\ c)$
$\wedge ([c1'],[c1'']) \in R \wedge ([c2'],[c2'']) \in R$
$\wedge\ b' \equiv_d b''$
by *force*

with *passump1 transR d-indistinguishable-trans*
have $([c1],[c1'']) \in R \wedge ([c2],[c2'']) \in R \wedge b \equiv_d b''$
by (*metis trans-def*)

with *passump1 passump2 R1-def* **have** $(V,V'') \in R1$
by *auto*
}
thus *?thesis* **unfolding** *trans-def* **by** *blast*
qed

have *transR2*: *trans R2*
proof −
{
fix $V\ V'\ V''$
assume *p1*: $(V,V') \in R2$
assume *p2*: $(V',V'') \in R2$

from *p1 R2-def* **obtain** $\iota\ \iota'\ b\ b'\ c1\ c1'$ **where** *passump1*:
$V = [while_{\iota}\ b\ do\ c1\ od]$
$\wedge\ V' = [while_{\iota'}\ b'\ do\ c1'\ od]$
$\wedge\ \iota \notin set\ (PPc\ c) \wedge \iota' \notin set\ (PPc\ c)$
$\wedge ([c1],[c1']) \in R \wedge b \equiv_d b'$
by *force*

with *p2 R2-def* **obtain** $\iota''\ b''\ c1''$ **where** *passump2*:
$V'' = [while_{\iota''}\ b''\ do\ c1''\ od] \wedge \iota'' \notin set\ (PPc\ c)$
$\wedge ([c1'],[c1'']) \in R \wedge b' \equiv_d b''$
by *force*

with *passump1 transR d-indistinguishable-trans*
have $([c1],[c1'']) \in R \wedge b \equiv_d b''$
by (*metis trans-def*)

with *passump1 passump2 R2-def* **have** $(V,V'') \in R2$
by *auto*
}
thus *?thesis* **unfolding** *trans-def* **by** *blast*
qed

from *SdlHPPR* **have** *eqlenR*: $\forall (V,V') \in R.\ length\ V = length\ V'$

by (*simp add*: *Strong-dlHPP-Bisimulation-def*)
from *R1-def eqlenR* **have** *eqlenR1*: $\forall (V,V')\in R1.$ *length* V = *length* V'
by *auto*
from *R2-def eqlenR* **have** *eqlenR2*: $\forall (V,V')\in R2.$ *length* V = *length* V'
by *auto*

from *R1-def R2-def* **have** *Domain R1* $\cap$ *Domain R2* = {}
by *auto*

with *commonArefl-subset-commonDomain* **have** *Arefl-a*:
Arefl R1 $\cap$ *Arefl R2* = {}
by *force*

with *symR1 symR2 transR1 transR2 eqlenR1 eqlenR2 trans-RuR'*
have *trans* (*R1* $\cup$ *R2*)
by *force*
with *R0-def* **show** *trans R0* **by** *auto*
next
fix V V'
assume *inR0*: $(V,V') \in R0$
with *R0-def R1-def R2-def* **show** *length* V = *length* V' **by** *auto*
next
fix V V' i
assume *inR0*: $(V,V') \in R0$
assume *irange*: $i <$ *length* V
assume *notIDC*: $\neg$ *IDC d* $(V!i)$
(*htchLoc* (*pp* $(V!i)$))

from *inR0 R0-def R1-def R2-def* **obtain** ι ι' b b' *c1 c1' c2 c2'*
where *R0pair*: ((V = [*c1*;(*while*$_\iota$ *b do c2 od*)]
$\wedge$ V' = [*c1'*;(*while*$_{\iota'}$ *b' do c2' od*)])
$\vee$ (V = [*while*$_\iota$ *b do c1 od*] $\wedge$ V' = [*while*$_{\iota'}$ *b' do c1' od*]))
$\wedge$ $\iota \notin$ *set* (*PPc c*) $\wedge$ $\iota' \notin$ *set* (*PPc c*)
$\wedge$ ([*c1*],[*c1'*]) $\in R$ $\wedge$ ([*c2*],[*c2'*]) $\in R$
$\wedge$ $b \equiv_d b'$
by *force*

with *irange SdlHPPR strongdlHPPB-NDCIDCaux*
[*of d PP R* [*c1*] [*c1'*] *i*]
have *c1NDCIDC*:
NDC d c1 $\vee$ *IDC d c1* (*htchLoc* (*pp c1*))
by *auto*

— first alternative for V and V'
have *case1*: *NDC d* (*c1*;(*while*$_\iota$ *b do c2 od*)) $\vee$
IDC d (*c1*;(*while*$_\iota$ *b do c2 od*))
(*htchLoc* (*pp* (*c1*;(*while*$_\iota$ *b do c2 od*))))
proof −
have *eqnextmem*: $\bigwedge m.$ $[\![$*c1*;(*while*$_\iota$ *b do c2 od*)$]\!](m) = [\![$*c1*$]\!](m)$

proof –
fix m
from *nextmem-exists-and-unique* **obtain** m' **where** *c1nextmem*:
$\exists p\ \alpha.\ \langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge (\forall m''.\ (\exists p\ \alpha.\ \langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m''\rangle) \longrightarrow m'' = m')$
by *force*

hence *eqdir1*: $[\![c1]\!](m) = m'$
by (*simp add*: *NextMem-def*, *auto*)

from *c1nextmem* **obtain** $p\ \alpha$ **where** $\langle c1,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
by *auto*

with *c1nextmem* **have**
$\exists p\ \alpha.\ \langle c1;(while_\iota\ b\ do\ c2\ od),m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge (\forall m''.\ (\exists p\ \alpha.\ \langle c1;(while_\iota\ b\ do\ c2\ od),m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m''\rangle)$
$\longrightarrow m'' = m')$
by (*auto*, *metis MWLsSteps-det.seq1 MWLsSteps-det.seq2*
option.exhaust, *metis MWLsSteps-det-cases*(*3*))

hence *eqdir2*: $[\![c1;(while_\iota\ b\ do\ c2\ od)]\!](m) = m'$
by (*simp add*: *NextMem-def*, *auto*)

with *eqdir1* **show** $[\![c1;(while_\iota\ b\ do\ c2\ od)]\!](m) = [\![c1]\!](m)$
by *auto*
qed

have *eqpp*: $pp\ (c1;(while_\iota\ b\ do\ c2\ od)) = pp\ c1$
by *simp*

with *c1NDCIDC eqnextmem* **show**
$NDC\ d\ (c1;(while_\iota\ b\ do\ c2\ od)) \vee$
$IDC\ d\ (c1;(while_\iota\ b\ do\ c2\ od))$
$(htchLoc\ (pp\ (c1;(while_\iota\ b\ do\ c2\ od))))$
by (*simp add*: *IDC-def NDC-def*)
qed

— second alternative for V V'
have *case2*: $NDC\ d\ (while_\iota\ b\ do\ c1\ od)$
proof –
{
fix m
from *nextmem-exists-and-unique* **obtain** $m'\ p\ \alpha$
where *whilenextmem*: $\langle while_\iota\ b\ do\ c1\ od,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m'\rangle$
$\wedge (\forall m''.\ (\exists p\ \alpha.\ \langle while_\iota\ b\ do\ c1\ od,m\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m''\rangle)$
$\longrightarrow m'' = m')$
by *blast*

hence $m = m'$

by (*metis MWLsSteps-det.whilefalse MWLsSteps-det.whiletrue*)

with *whilenextmem* **have** *eqnextmem*:
$[\![while_\iota\ b\ do\ c1\ od]\!](m) = m$
by (*simp add*: *NextMem-def*, *auto*)
}
thus *NDC d* ($while_\iota$ *b do c1 od*)
by (*simp add*: *NDC-def*)
qed

from *R0pair case1 case2 irange notIDC*
show *NDC d* ($V!i$)
by *force*
next
fix $V\ V'\ i\ m1\ m1'\ m2\ \alpha\ p$
assume *inR0*: $(V,V') \in R0$
assume *irange*: $i < length\ V$
assume *step*: $\langle V!i,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle p,m2\rangle$
assume *dhequal*: $m1 \sim_{d,htchLocSet\ PP} m1'$

from *inR0 R0-def R1-def R2-def* **obtain** $\iota\ \iota'\ b\ b'\ c1\ c1'\ c2\ c2'$
where *R0pair*: $((V = [c1;(while_\iota\ b\ do\ c2\ od)]$
$\wedge\ V' = [c1';(while_{\iota'}\ b'\ do\ c2'\ od)])$
$\vee\ (V = [while_\iota\ b\ do\ c1\ od] \wedge V' = [while_{\iota'}\ b'\ do\ c1'\ od]))$
$\wedge\ \iota \notin set\ (PPc\ c) \wedge \iota' \notin set\ (PPc\ c)$
$\wedge\ ([c1],[c1']) \in R \wedge ([c2],[c2']) \in R$
$\wedge\ b \equiv_d b'$
by *force*

— Case 1: V and V' are sequential commands
have *case1*: $[\![\ V = [c1;(while_\iota\ b\ do\ c2\ od)];$
$V' = [c1';(while_{\iota'}\ b'\ do\ c2'\ od)]\ ]\!]$
$\Longrightarrow \exists\, p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle\ \wedge$
stepResultsinR $p\ p'\ (R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R)\ \wedge$
dhequality-alternative $d\ PP\ (pp\ (V!i))\ m2\ m2'$
proof −
assume *Vassump*: $V = [c1;(while_\iota\ b\ do\ c2\ od)]$
assume *V'assump*: $V' = [c1';(while_{\iota'}\ b'\ do\ c2'\ od)]$

have *eqpp*: $pp\ (c1;(while_\iota\ b\ do\ c2\ od)) = pp\ c1$
by *simp*

from *Vassump irange step irange* **obtain** *c3*
where *case-distinction*:
$(p = Some\ (while_\iota\ b\ do\ c2\ od) \wedge \langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle None,m2\rangle)$
$\vee\ (p = Some\ (c3;(while_\iota\ b\ do\ c2\ od))$
$\wedge\ \langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ c3,m2\rangle)$
by (*simp*, *metis MWLsSteps-det-cases*(*3*))
moreover

— Case 1a: first command terminates
{
assume *passump*: $p = Some\ (while_\iota\ b\ do\ c2\ od)$
assume *stepassump*: $\langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle None,m2\rangle$

with *R0pair SdlHPPR dhequal*
strongdlHPPB-aux[*of d PP R*
- [*c1*] [*c1'*] *m1* α *None m2 m1'*]
obtain $p'\ \alpha'\ m2'$ **where** *c1c1'reason*:
$p' = None \wedge \langle c1',m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge (\alpha,\alpha') \in R \wedge$
dhequality-alternative d PP (*pp c1*) *m2 m2'*
by (*simp add*: *stepResultsinR-def*, *fastforce*)

hence *conclpart*:
$\langle c1';(while_{\iota'}\ b'\ do\ c2'\ od),m1'\rangle$
$\rightarrow\triangleleft\alpha'\triangleright \langle Some\ (while_{\iota'}\ b'\ do\ c2'\ od),m2'\rangle$
by (*auto*, *simp add*: *MWLsSteps-det.seq1*)

from *R0pair R0-def R2-def* **have**
$([while_\iota\ b\ do\ c2\ od],[while_{\iota'}\ b'\ do\ c2'\ od]) \in R0$
by *simp*

with *passump V'assump Vassump eqpp conclpart*
c1c1'reason irange
have $\exists\, p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
stepResultsinR $p\ p'\ (R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee$
$(\alpha,\alpha') \in R) \wedge$
dhequality-alternative d PP (*pp* $(V!i)$) *m2 m2'*
by (*simp add*: *stepResultsinR-def*, *auto*)
}
moreover
— Case 1b: first command does not terminate
{
assume *passump*: $p = Some\ (c3;(while_\iota\ b\ do\ c2\ od))$
assume *stepassump*: $\langle c1,m1\rangle \rightarrow\triangleleft\alpha\triangleright \langle Some\ c3,m2\rangle$

with *R0pair SdlHPPR dhequal*
strongdlHPPB-aux[*of d PP R*
- [*c1*] [*c1'*] *m1* α *Some c3 m2 m1'*]
obtain $p'\ c3'\ \alpha'\ m2'$ **where** *c1c1'reason*:
$p' = Some\ c3' \wedge \langle c1',m1'\rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2'\rangle \wedge$
$([c3],[c3']) \in R \wedge (\alpha,\alpha') \in R \wedge$
dhequality-alternative d PP (*pp c1*) *m2 m2'*
by (*simp add*: *stepResultsinR-def*, *fastforce*)

hence *conclpart*:
$\langle c1';(while_{\iota'}\ b'\ do\ c2'\ od),m1'\rangle \rightarrow\triangleleft\alpha'\triangleright$
$\langle Some\ (c3';while_{\iota'}\ b'\ do\ c2'\ od),m2'\rangle$
by (*auto*, *simp add*: *MWLsSteps-det.seq2*)

from *c1c1′reason R0pair R0-def R1-def* **have**
$([c3;while_\iota\ b\ do\ c2\ od],[c3';while_{\iota'}\ b'\ do\ c2'\ od]) \in R0$
by *simp*

with *passump V′assump Vassump eqpp conclpart c1c1′reason irange*
have $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R0 \cup R) \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
by (*simp add*: *stepResultsinR-def*, *auto*)
}
ultimately show $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R0 \cup R) \wedge$
$((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
by *auto*
qed

— Case 2: V and V' are while commands
have *case2*: $[\![\ V = [while_\iota\ b\ do\ c1\ od];\ V' = [while_{\iota'}\ b'\ do\ c1'\ od]\ ]\!]$
$\Longrightarrow \exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle \wedge$
$stepResultsinR\ p\ p'\ (R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
$dhequality\text{-}alternative\ d\ PP\ (pp\ (V!i))\ m2\ m2'$
proof −
assume *Vassump*: $V = [while_\iota\ b\ do\ c1\ od]$
assume *V′assump*: $V' = [while_{\iota'}\ b'\ do\ c1'\ od]$

from *Vassump irange step* **have** *case-distinction*:
$(p = None \wedge BMap\ (E\ b\ m1) = False)$
$\vee\ p = (Some\ (c1;while_\iota\ b\ do\ c1\ od)) \wedge BMap\ (E\ b\ m1) = True$
by (*simp*, *metis MWLsSteps-det-cases*(5) *option.simps*(2))
moreover
— Case 2a: b evaluates to False
{
assume *passump*: $p = None$
assume *eval*: $BMap\ (E\ b\ m1) = False$

with *Vassump step irange* **have** *stepconcl*: $\alpha = [] \wedge m2 = m1$
by (*simp*, *metis* (*no-types*) *MWLsSteps-det-cases*(5))

from *eval R0pair dhequal* **have** *eval′*: $BMap\ (E\ b'\ m1') = False$
by (*simp add*: *d-indistinguishable-def dH-equal-def*, *auto*)

hence $\langle while_{\iota'}\ b'\ do\ c1'\ od,m1'\rangle \rightarrow\lhd[]\rhd\ \langle None,m1'\rangle$
by (*simp add*: *MWLsSteps-det.whilefalse*)

with *passump R-def Vassump V′assump stepconcl dhequal irange*
have $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow\lhd\alpha'\rhd\ \langle p',m2'\rangle \wedge$

stepResultsinR p p′ $(R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
dhequality-alternative d PP (*pp* (*V*!*i*)) *m2 m2′*
by (*simp add*: *stepResultsinR-def dhequality-alternative-def*,
auto)
}
moreover
— Case 2b: b evaluates to True
{
assume *passump*: $p = (Some\ (c1;while_\iota\ b\ do\ c1\ od))$
assume *eval*: $BMap\ (E\ b\ m1) = True$

with *Vassump step irange* **have** *stepconcl*: $\alpha = [] \wedge m2 = m1$
by (*simp*, *metis* (*no-types*) *MWLsSteps-det-cases*(*5*))

from *eval R0pair dhequal* **have** *eval′*:
$BMap\ (E\ b'\ m1') = True$
by (*simp add*: *d-indistinguishable-def dH-equal-def*, *auto*)

hence *step′*: $\langle while_{\iota'}\ b'\ do\ c1'\ od,m1' \rangle \rightarrow\triangleleft[]\triangleright$
$\langle Some\ (c1';while_{\iota'}\ b'\ do\ c1'\ od),m1' \rangle$
by (*simp add*: *MWLsSteps-det.whiletrue*)

from *R1-def R0pair* **have** *inR1*:
$([c1;while_\iota\ b\ do\ c1\ od],[c1';while_{\iota'}\ b'\ do\ c1'\ od]) \in R1$
by *auto*

with *step′ R0-def passump R-def Vassump V′assump*
stepconcl dhequal irange
have $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1' \rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2' \rangle \wedge$
stepResultsinR p p′ $(R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
dhequality-alternative d PP (*pp* (*V*!*i*)) *m2 m2′*
by (*simp add*: *stepResultsinR-def dhequality-alternative-def*,
auto)
}
ultimately
show $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1' \rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2' \rangle \wedge$
stepResultsinR p p′ $(R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
dhequality-alternative d PP (*pp* (*V*!*i*)) *m2 m2′*
by *auto*
qed

with *case1 R0pair* **show** $\exists p'\ \alpha'\ m2'.\ \langle V'!i,m1' \rangle \rightarrow\triangleleft\alpha'\triangleright \langle p',m2' \rangle \wedge$
stepResultsinR p p′ $(R0 \cup R) \wedge ((\alpha,\alpha') \in R0 \vee (\alpha,\alpha') \in R) \wedge$
dhequality-alternative d PP (*pp* (*V*!*i*)) *m2 m2′*
by *auto*
qed

with *Areflassump Up-To-Technique*
have *SdlHPPB d PP* $(R0 \cup R)$

by *auto*

with *inR2 R0-def* **show** $\exists R.$ *SdlHPPB d PP R* $\wedge$
([*while*$_\iota$ *b do c od*], [*while*$_\iota$ *b do c od*]) $\in R$
by *auto*
qed

end

end

4 Security type system

4.1 Abstract security type system with soundness proof

We formalize an abstract version of the type system in [2] using locales [1]. Our formalization of the type system is abstract in the sense that the rules specify abstract semantic side conditions on the expressions within a command that satisfy for proving the soundness of the rules. That is, it can be instantiated with different syntactic approximations for these semantic side conditions in order to achieve a type system for a concrete language for Boolean and arithmetic expressions. Obtaining a soundness proof for such a concrete type system then boils down to proving that the concrete type system interprets the abstract type system.

We prove the soundness of the abstract type system by simply applying the compositionality results proven before.

theory *Type-System*
imports *Language-Composition*
begin

locale *Type-System* =
WWP: *WHATWHERE-Secure-Programs E BMap DA lH*
for *E* :: ($'exp$, $'id$, $'val$) *Evalfunction*
and *BMap* :: $'val \Rightarrow bool$
and *DA* :: ($'id$, $'d$::*order*) *DomainAssignment*
and *lH* :: ($'d$, $'exp$) *lHatches*
+
fixes
AssignSideCondition :: $'id \Rightarrow {}'exp \Rightarrow nat \Rightarrow bool$
and *WhileSideCondition* :: $'exp \Rightarrow bool$
and *IfSideCondition* ::
$'exp \Rightarrow ('exp,'id)$ *MWLsCom* $\Rightarrow ('exp,'id)$ *MWLsCom* $\Rightarrow bool$
assumes *semAssignSC*: *AssignSideCondition x e* $\iota \Longrightarrow$
$e \equiv_{DA\ x,(htchLoc\ \iota)} e \wedge (\forall m\ m'\ d\ \iota'.\ (m \sim_{d,(htchLoc\ \iota')} m' \wedge$

$[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m'))$
$\longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m'))$
and *semWhileSC*: $WhileSideCondition\ e \Longrightarrow \forall\, d.\ e \equiv_d e$
and *semIfSC*: $IfSideCondition\ e\ c1\ c2 \Longrightarrow \forall\, d.\ e \equiv_d e$
begin

— Security typing rules for the language commands
inductive
$ComSecTyping :: ('exp,\ 'id)\ MWLsCom \Rightarrow bool$
$(\vdash_{\mathcal{C}}$ -)
and $ComSecTypingL :: ('exp,'id)\ MWLsCom\ list \Rightarrow bool$
$(\vdash_{\mathcal{V}}$ -)
where
Skip: $\vdash_{\mathcal{C}} skip_\iota$ |
Assign: $[\![\ AssignSideCondition\ x\ e\ \iota\ ]\!] \Longrightarrow \vdash_{\mathcal{C}} x :=_\iota e$ |
Spawn: $[\![\ \vdash_{\mathcal{V}} V\ ]\!] \Longrightarrow \vdash_{\mathcal{C}} spawn_\iota\ V$ |
Seq: $[\![\ \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2\ ]\!] \Longrightarrow \vdash_{\mathcal{C}} c1;c2$ |
While: $[\![\ \vdash_{\mathcal{C}} c;\ WhileSideCondition\ b\ ]\!]$
$\Longrightarrow \vdash_{\mathcal{C}} while_\iota\ b\ do\ c\ od$ |
If: $[\![\ \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2;\ IfSideCondition\ b\ c1\ c2\ ]\!]$
$\Longrightarrow \vdash_{\mathcal{C}} if_\iota\ b\ then\ c1\ else\ c2\ fi$ |
Parallel: $[\![\ \forall\, i < length\ V.\ \vdash_{\mathcal{C}} V!i\ ]\!] \Longrightarrow \vdash_{\mathcal{V}} V$

inductive-cases *parallel-cases*:
$\vdash_{\mathcal{V}} V$

definition *auxiliary-predicate*
where
auxiliary-predicate $V \equiv$ *unique-PPV* $V \longrightarrow$ *WHATWHERE-Secure* V

— soundness proof of abstract type system
theorem *ComSecTyping-single-is-sound*:
$[\![\ \vdash_{\mathcal{C}} c;$ *unique-PPc* $c\ ]\!]$
$\Longrightarrow$ *WHATWHERE-Secure* $[c]$
proof (*induct rule*: *ComSecTyping-ComSecTypingL.inducts*(*1*)
[*of - - auxiliary-predicate*], *simp-all add*: *auxiliary-predicate-def*)
fix ι
show *WHATWHERE-Secure* $[skip_\iota]$
by (*metis WHATWHERE-Secure-Skip*)
next
fix $x\ e\ \iota$
assume *AssignSideCondition* $x\ e\ \iota$
thus *WHATWHERE-Secure* $[x :=_\iota e]$
by (*metis WHATWHERE-Secure-Assign semAssignSC*)
next
fix $V\ \iota$
assume *IH*: *unique-PPV* $V \longrightarrow$ *WHATWHERE-Secure* V
assume *uniPPspawn*: *unique-PPc* $(spawn_\iota\ V)$
hence *unique-PPV* V

by (*simp add*: *unique-PPV-def unique-PPc-def*)
with *IH* **have** *WHATWHERE-Secure V*
..
with *uniPPspawn* **show** *WHATWHERE-Secure* [$spawn_\iota$ *V*]
by (*metis Compositionality-Spawn*)
next
fix *c1 c2*
assume *IH1*: *unique-PPc c1* $\Longrightarrow$ *WHATWHERE-Secure* [*c1*]
assume *IH2*: *unique-PPc c2* $\Longrightarrow$ *WHATWHERE-Secure* [*c2*]
assume *uniPPc1c2*: *unique-PPc* (*c1*;*c2*)
from *uniPPc1c2* **have** *uniPPc1*: *unique-PPc c1*
by (*simp add*: *unique-PPc-def*)
with *IH1* **have** *IS1*: *WHATWHERE-Secure* [*c1*]
.
from *uniPPc1c2* **have** *uniPPc2*: *unique-PPc c2*
by (*simp add*: *unique-PPc-def*)
with *IH2* **have** *IS2*: *WHATWHERE-Secure* [*c2*]
.

from *IS1 IS2 uniPPc1c2* **show** *WHATWHERE-Secure* [*c1*;*c2*]
by (*metis Compositionality-Seq*)
next
fix *c b* ι
assume *SC*: *WhileSideCondition b*
assume *IH*: *unique-PPc c* $\Longrightarrow$ *WHATWHERE-Secure* [*c*]
assume *uniPPwhile*: *unique-PPc* ($while_\iota$ *b do c od*)
hence *unique-PPc c*
by (*simp add*: *unique-PPc-def*)
with *IH* **have** *WHATWHERE-Secure* [*c*]
.
with *uniPPwhile SC* **show** *WHATWHERE-Secure* [$while_\iota$ *b do c od*]
by (*metis Compositionality-While semWhileSC*)
next
fix *c1 c2 b* ι
assume *SC*: *IfSideCondition b c1 c2*
assume *IH1*: *unique-PPc c1* $\Longrightarrow$ *WHATWHERE-Secure* [*c1*]
assume *IH2*: *unique-PPc c2* $\Longrightarrow$ *WHATWHERE-Secure* [*c2*]
assume *uniPPif*: *unique-PPc* (if_ι *b then c1 else c2 fi*)
from *uniPPif* **have** *unique-PPc c1*
by (*simp add*: *unique-PPc-def*)
with *IH1* **have** *IS1*: *WHATWHERE-Secure* [*c1*]
.
from *uniPPif* **have** *unique-PPc c2*
by (*simp add*: *unique-PPc-def*)
with *IH2* **have** *IS2*: *WHATWHERE-Secure* [*c2*]
.
from *IS1 IS2 SC uniPPif* **show**
WHATWHERE-Secure [if_ι *b then c1 else c2 fi*]
by (*metis Compositionality-If semIfSC*)

next
fix V
assume IH: $\forall\, i < \mathit{length}\ V.\ \vdash_{\mathcal{C}}\ V\ !\ i\ \wedge$
$(\mathit{unique\text{-}PPc}\ (V!i) \longrightarrow \mathit{WHATWHERE\text{-}Secure}\ [V!i])$
have $\mathit{unique\text{-}PPV}\ V \longrightarrow (\forall\, i < \mathit{length}\ V.\ \mathit{unique\text{-}PPc}\ (V!i))$
by (*metis uniPPV-uniPPc*)
with *IH* **have** $\mathit{unique\text{-}PPV}\ V \longrightarrow (\forall\, i < \mathit{length}\ V.\ \mathit{WHATWHERE\text{-}Secure}\ [V!i])$

by *auto*
thus *uniPPV*: $\mathit{unique\text{-}PPV}\ V \longrightarrow \mathit{WHATWHERE\text{-}Secure}\ V$
by (*metis parallel-composition*)
qed

theorem *ComSecTyping-list-is-sound*:
$[\![\ \vdash_{\mathcal{V}}\ V;\ \mathit{unique\text{-}PPV}\ V\ ]\!] \Longrightarrow \mathit{WHATWHERE\text{-}Secure}\ V$
by (*metis ComSecTyping-single-is-sound parallel-cases parallel-composition uniPPV-uniPPc*)

end

end

4.2 Example language for Boolean and arithmetic expressions

As and example, we provide a simple example language for instantiating the parameter *'exp* for the language for Boolean and arithmetic expressions (from the entry Strong-Security).

theory *Expr*
imports *Types*
begin

— type parameters:
— 'val: numbers, boolean constants....
— 'id: identifier names

type-synonym $('val)\ \mathit{operation} = 'val\ \mathit{list} \Rightarrow 'val$

datatype $('id, 'val)\ \mathit{Expr} =$
$\mathit{Const}\ 'val\ |$
$\mathit{Var}\ 'id\ |$
$\mathit{Op}\ 'val\ \mathit{operation}\ (('id, 'val)\ \mathit{Expr})\ \mathit{list}$

— defining a simple recursive evaluation function on this datatype

primrec *ExprEval* :: $(('id, 'val)$ *Expr*, $'id, 'val)$ *Evalfunction*
and *ExprEvalL* :: $(('id, 'val)$ *Expr*) *list* $\Rightarrow$ $('id, 'val)$ *State* $\Rightarrow$ $'val$ *list*
where
ExprEval (*Const v*) $m = v$ |
ExprEval (*Var x*) $m = (m\ x)$ |
ExprEval (*Op f arglist*) $m = (f$ (*ExprEvalL arglist m*)) |

ExprEvalL [] m = [] |
ExprEvalL ($e\#V$) m = (*ExprEval e m*)#(*ExprEvalL V m*)

end

4.3 Example interpretation of abstract security type system

Using the example instantiation of the language for Boolean and arithmetic expressions, we give an example instantiation of our abstract security type system, instantiating the parameter for domains *'d* with a two-level security lattice (from the entry Strong-Security).

theory *Domain-example*
imports *Expr*
begin

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

datatype *Dom* = *low* | *high*

instantiation *Dom* :: *order*
begin

definition
less-eq-Dom-def: $d1 \leq d2 = ($*if* $d1 = d2$ *then True*
else (*if* $d1 = low$ *then True else False*))

definition
less-Dom-def: $d1 < d2 = ($*if* $d1 = d2$ *then False*
else (*if* $d1 = low$ *then True else False*))

instance proof
fix *x y z* :: *Dom*
show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
unfolding *less-eq-Dom-def less-Dom-def* **by** *auto*
show $x \leq x$ **unfolding** *less-eq-Dom-def* **by** *auto*
show $[\![x \leq y;\ y \leq z]\!] \Longrightarrow x \leq z$
unfolding *less-eq-Dom-def* **by** ((*split split-if-asm*)+, *auto*)

show $[\![x \le y;\ y \le x]\!] \Longrightarrow x = y$
  unfolding *less-eq-Dom-def* **by** ((*split split-if-asm*)+,
    auto, (*split split-if-asm*)+, *auto*)
qed

end

end

theory *Type-System-example*
imports *Type-System ../Strong−Security/Expr ../Strong−Security/Domain-example*
begin

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

consts *DA* :: ('*id*,*Dom*) *DomainAssignment*
consts *BMap* :: '*val* ⇒ *bool*
consts *lH* :: (*Dom*,('*id*,'*val*) *Expr*) *lHatches*

— redefine all the abbreviations necessary for auxiliary lemmas with the correct parameter instantiation

abbreviation *MWLsStepsdet'* ::
  (('*id*,'*val*) *Expr*, '*id*, '*val*, (('*id*,'*val*) *Expr*,'*id*) *MWLsCom*) *TLSteps-curry*
((*1*⟨-,/-⟩) →◁-▷/ (*1*⟨-,/-⟩) [*0*,*0*,*0*,*0*,*0*] *81*)
where
⟨*c1*,*m1*⟩ →◁α▷ ⟨*c2*,*m2*⟩ ≡
  ((*c1*,*m1*),α,(*c2*,*m2*)) ∈ *MWLs-semantics.MWLsSteps-det ExprEval BMap*

abbreviation *d-equal'* :: ('*id*, '*val*) *State*
  ⇒ *Dom* ⇒ ('*id*, '*val*) *State* ⇒ *bool*
((- $=_{-}$ -))
where
$m =_d m'$ ≡ *WHATWHERE.d-equal DA d m m'*

abbreviation *dH-equal'* :: ('*id*, '*val*) *State* ⇒ *Dom*
  ⇒ (*Dom*,('*id*,'*val*) *Expr*) *Hatches*
  ⇒ ('*id*, '*val*) *State* ⇒ *bool*
((- $\sim_{-,-}$ -))
where
$m \sim_{d,H} m'$ ≡ *WHATWHERE.dH-equal ExprEval DA d H m m'*

abbreviation *NextMem'* :: (('*id*,'*val*) *Expr*, '*id*) *MWLsCom*
  ⇒ ('*id*,'*val*) *State* ⇒ ('*id*,'*val*) *State*
([\![-]\!]'(-'))
where

$[\![c]\!](m)$
 ≡ *WHATWHERE.NextMem* (*MWLs-semantics.MWLsSteps-det ExprEval BMap*)
c m

abbreviation *dH-indistinguishable′* :: (*′id*,*′val*) *Expr* ⇒ *Dom*
 ⇒ (*Dom*,(*′id*,*′val*) *Expr*) *Hatches* ⇒ (*′id*,*′val*) *Expr* ⇒ *bool*
((- $\equiv_{-,-}$ -))
where
e1 $\equiv_{d,H}$ *e2*
 ≡ *WHATWHERE-Secure-Programs.dH-indistinguishable ExprEval DA d H e1 e2*

abbreviation *htchLoc* :: *nat* ⇒ (*Dom*, (*′id*,*′val*) *Expr*) *Hatches*
where
htchLoc ι ≡ *WHATWHERE.htchLoc lH* ι

— Security typing rules for expressions
inductive
ExprSecTyping :: (*Dom*, (*′id*,*′val*) *Expr*) *Hatches*
 ⇒ (*′id*, *′val*) *Expr* ⇒ *Dom* ⇒ *bool*
(- $\vdash_{\mathcal{E}}$ - : -)
for *H* :: (*Dom*, (*′id*, *′val*) *Expr*) *Hatches*
where
Consts: *H* $\vdash_{\mathcal{E}}$ (*Const v*) : *d* |
Vars: *DA x* = *d* ⟹ *H* $\vdash_{\mathcal{E}}$ (*Var x*) : *d* |
Hatch: (*d*,*e*) ∈ *H* ⟹ *H* $\vdash_{\mathcal{E}}$ *e* : *d* |
Ops: ⟦ ∀ *i* < *length arglist*. *H* $\vdash_{\mathcal{E}}$ (*arglist*!*i*) : (*dl*!*i*) ∧ (*dl*!*i*) ≤ *d* ⟧
 ⟹ *H* $\vdash_{\mathcal{E}}$ (*Op f arglist*) : *d*

— function substituting a certain expression with another expression in expressions

primrec *Subst* :: (*′id*, *′val*) *Expr* ⇒ (*′id*, *′val*) *Expr*
 ⇒ (*′id*, *′val*) *Expr* ⇒ (*′id*, *′val*) *Expr*
(-<-\->)
and *SubstL* :: (*′id*, *′val*) *Expr list* ⇒ (*′id*, *′val*) *Expr*
 ⇒ (*′id*, *′val*) *Expr* ⇒ (*′id*, *′val*) *Expr list*
where
(*Const v*)<*e1**e2*> = (*if e1*=(*Const v*) *then e2 else* (*Const v*)) |
(*Var x*)<*e1**e2*> = (*if e1*=(*Var x*) *then e2 else* (*Var x*)) |
(*Op f arglist*)<*e1**e2*> = (*if e1*=(*Op f arglist*) *then e2 else*
 (*Op f* (*SubstL arglist e1 e2*))) |

SubstL [] *e1 e2* = [] |
SubstL (*e*#*V*) *e1 e2* = (*e*<*e1**e2*>)#(*SubstL V e1 e2*)

definition *SubstClosure* :: *′id* ⇒ (*′id*, *′val*) *Expr* ⇒ *bool*
where
SubstClosure x e ≡ ∀ (*d′*,*e′*,ι') ∈ *lH*. (*d′*,*e′*<(*Var x*)*e*>,ι') ∈ *lH*

definition *synAssignSC* :: $'id \Rightarrow ('id, 'val)$ *Expr* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
synAssignSC $x\ e\ \iota \equiv \exists\, d.\ ((htchLoc\ \iota) \vdash_{\mathcal{E}} e : d \wedge d \leq DA\ x)$
$\wedge$ (*SubstClosure* $x\ e$)

definition *synWhileSC* :: $('id, 'val)$ *Expr* $\Rightarrow$ *bool*
where
synWhileSC $e \equiv (\exists\, d.\ (\{\} \vdash_{\mathcal{E}} e : d) \wedge (\forall\, d'.\ d \leq d'))$

definition *synIfSC* :: $('id, 'val)$ *Expr*
$\Rightarrow (('id, 'val)$ *Expr*, $'id)$ *MWLsCom*
$\Rightarrow (('id, 'val)$ *Expr*, $'id)$ *MWLsCom* $\Rightarrow$ *bool*
where
synIfSC $e\ c1\ c2 \equiv \exists\, d.\ (\{\} \vdash_{\mathcal{E}} e : d \wedge (\forall\, d'.\ d \leq d'))$

— auxiliary lemma for locale interpretation (theorem 7 in original paper)
lemma *ExprTypable-with-smallerd-implies-dH-indistinguishable*:
$[\![\ H \vdash_{\mathcal{E}} e : d';\ d' \leq d\ ]\!] \Longrightarrow e \equiv_{d,H} e$
proof (*induct rule*: *ExprSecTyping.induct*,
simp-all add: *WHATWHERE-Secure-Programs.dH-indistinguishable-def*
WHATWHERE.dH-equal-def WHATWHERE.d-equal-def, *auto*)
fix *dl arglist f m1 m2* d' d
assume *main*: $\forall\, i <$ *length arglist*.
$(H \vdash_{\mathcal{E}} (arglist!i) : (dl!i)) \wedge (dl!i \leq d \longrightarrow$
$(\forall\, m1\ m2.\ (\forall\, x.\ DA\ x \leq d \longrightarrow m1\ x = m2\ x) \wedge$
$(\forall\, (d',e) {\in} H.\ d' \leq d \longrightarrow ExprEval\ e\ m1 = ExprEval\ e\ m2) \longrightarrow$
$ExprEval\ (arglist!i)\ m1 = ExprEval\ (arglist!i)\ m2)) \wedge dl!i \leq d'$
assume *smaller*: $d' \leq d$
assume *eqeval*: $\forall\, (d',e) \in H.\ d' \leq d \longrightarrow ExprEval\ e\ m1 = ExprEval\ e\ m2$
assume *eqstate*: $\forall\, x.\ DA\ x \leq d \longrightarrow m1\ x = m2\ x$

from *main smaller* **have** *irangesubst*:
$\forall\, i <$ *length arglist*. $dl!i \leq d$
by (*metis order-trans*)

with *eqstate eqeval main* **have**
$\forall\, i <$ *length arglist*. *ExprEval* (*arglist*!i) *m1*
= *ExprEval* (*arglist*!i) *m2*
by *force*

hence *substmap*: (*ExprEvalL arglist m1*) = (*ExprEvalL arglist m2*)
by (*induct arglist*, *auto*, *force*)

show f (*ExprEvalL arglist m1*) = f (*ExprEvalL arglist m2*)
by (*subst substmap*, *auto*)

qed

— auxiliary lemma about substitutions in expressions and in memories
lemma *substexp-substmem*:
$ExprEval\ e'\!<\!Var\ x\backslash e\!>\ m = ExprEval\ e'\ (m(x := ExprEval\ e\ m))$
$\wedge\ ExprEvalL\ (SubstL\ elist\ (Var\ x)\ e)\ m$
$=\ ExprEvalL\ elist\ (m(x := ExprEval\ e\ m))$
by (*induct-tac* e' **and** *elist*, *simp-all*)

— another auxiliary lemma for locale interpretation (lemma 8 in original paper)
lemma *SubstClosure-implications*:
$[\![\ SubstClosure\ x\ e;\ m \sim_{d,(htchLoc\ \iota')}\ m';$
$[\![x :=_\iota e]\!](m) =_d [\![x :=_\iota e]\!](m')\ ]\!]$
$\Longrightarrow [\![x :=_\iota e]\!](m) \sim_{d,(htchLoc\ \iota')} [\![x :=_\iota e]\!](m')$
proof −
fix $m1\ m1'$
assume *substclosure*: $SubstClosure\ x\ e$
assume *dequalm2*: $[\![x :=_\iota e]\!](m1) =_d [\![x :=_\iota e]\!](m1')$
assume *dhequalm1*: $m1 \sim_{d,(htchLoc\ \iota')} m1'$

from *MWLs-semantics.nextmem-exists-and-unique* **obtain** $m2$ **where** *m1step*:
$(\exists\, p\ \alpha.\ \langle x :=_\iota e, m1\rangle \rightarrow\!\lhd\alpha\rhd\ \langle p, m2\rangle)$
$\wedge\ (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle x :=_\iota e, m1\rangle \rightarrow\!\lhd\alpha\rhd\ \langle p, m''\rangle) \longrightarrow m'' = m2)$
by *force*
hence *m2-is-next*: $[\![x :=_\iota e]\!](m1) = m2$
by (*simp add*: *WHATWHERE.NextMem-def*, *auto*)
from *m1step* *MWLs-semantics.MWLsSteps-det.assign*
[*of ExprEval e m1 - x* ι *BMap*]
have *m2eq*: $m2 = m1(x := (ExprEval\ e\ m1))$
by *auto*

from *MWLs-semantics.nextmem-exists-and-unique* **obtain** $m2'$ **where** *m1'step*:
$(\exists\, p\ \alpha.\ \langle x :=_\iota e, m1'\rangle \rightarrow\!\lhd\alpha\rhd\ \langle p, m2'\rangle)$
$\wedge\ (\forall\, m''.\ (\exists\, p\ \alpha.\ \langle x :=_\iota e, m1'\rangle \rightarrow\!\lhd\alpha\rhd\ \langle p, m''\rangle) \longrightarrow m'' = m2')$
by *force*
hence *m2'-is-next*: $[\![x :=_\iota e]\!](m1') = m2'$
by (*simp add*: *WHATWHERE.NextMem-def*, *auto*)
from *m1'step* *MWLs-semantics.MWLsSteps-det.assign*
[*of ExprEval e m1' - x* ι *BMap*]
have *m2'eq*: $m2' = m1'(x := (ExprEval\ e\ m1'))$
by *auto*

from *m2eq* *substexp-substmem*
have *substeval1*: $\forall\, e'.\ ExprEval\ (e'\!<\!Var\ x\backslash e\!>)\ m1 = ExprEval\ e'\ m2$
by *force*

from *m2'eq* *substexp-substmem*
have *substeval2*: $\forall\, e'.\ ExprEval\ e'\!<\!Var\ x\backslash e\!>\ m1' = ExprEval\ e'\ m2'$
by *force*

from *substclosure* **have**
$\forall (d',e') \in htchLoc\ \iota'.\ (d',e'\!<\!Var\ x\backslash e\!>) \in (htchLoc\ \iota')$
by (*simp add*: *SubstClosure-def WHATWHERE.htchLoc-def*, *auto*)

with *dhequalm1* **have**
$\forall (d',e') \in htchLoc\ \iota'.$
$d' \leq d \longrightarrow ExprEval\ e'\!<\!Var\ x\backslash e\!>\ m1 = ExprEval\ e'\!<\!Var\ x\backslash e\!>\ m1'$
by (*simp add*: *WHATWHERE.dH-equal-def*, *auto*)

with *substeval1 substeval2* **have**
$\forall (d',e') \in htchLoc\ \iota'.$
$d' \leq d \longrightarrow ExprEval\ e'\ m2 = ExprEval\ e'\ m2'$
by *auto*

with *dequalm2 m2-is-next m2'-is-next*
show $[\![x :=_\iota e]\!](m1) \sim_{d,htchLoc\ \iota'} [\![x :=_\iota e]\!](m1')$
by (*simp add*: *WHATWHERE.dH-equal-def*)
qed

— interpretation of the abstract type system using the above definitions for the side conditions
interpretation *Type-System-example*: *Type-System ExprEval BMap DA lH synAssignSC synWhileSC synIfSC*
by (*unfold-locales*, *auto*,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable synAssignSC-def,
metis SubstClosure-implications synAssignSC-def,
simp add: *synWhileSC-def*,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable,

simp add: *synIfSC-def*,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable)

end

References

[1] C. Ballarin. Locales and Locale Expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[2] A. Lux, H. Mantel, and M. Perner. Scheduler-independent declassification. In *Proceedings of the 11th International Conference on Mathemat-*

ics of Program Construction (MPC), LNCS 7342, pages 25–47. Springer, 2012.